MITIGATING THE PERFORMANCE IMPACT OF MEMORY BLOAT

A Thesis
Presented to
The Academic Faculty

by

Sangho Lee

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science, College of Computing

Georgia Institute of Technology
December 2015

MITIGATING THE PERFORMANCE IMPACT OF MEMORY BLOAT

Approved by:

Dr. Santosh Pande, Advisor
School of Computer Science
Georgia Institute of Technology

Dr. Karsten Schwan
School of Computer Science
Georgia Institute of Technology

Dr. Alessandro Orso
School of Computer Science
Georgia Institute of Technology

Dr. Sudhakar Yalamanchili
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Hyesoon Kim
School of Computer Science
Georgia Institute of Technology

Date Approved: 12 August 2015

To my family,

ACKNOWLEDGEMENTS

First and foremost, I would sincerely like to thank my advisor, Dr. Santosh Pande, for his guidance and funding throughout my doctoral studies. Without his enthusiasm for research and intensive mentoring, I would not have been able to complete my doctoral studies successfully.

I would also like to thank all the members of my thesis committee: Dr. Alessandro Orso, Dr. Hyesoon Kim, Dr. Karsten Schwan, and Dr. Sudhakar Yalamanchili. Their comments, suggestions, and feedback were greatly helpful in shaping this thesis. I am especially grateful to Dr. Hyesoon Kim and Dr. Sudhakar Yalamanchili for providing partial financial support for two semesters.

Also, I would like to thank my previous internship hosts: Vinod Tipparaju, Susan Brownhill, Chihong Zhang, Teresa Johnson, and Easwaran Raman. The knowledge I gained through the internships was very useful in my research.

Former and current members in the lab also deserve my thanks. Romain, Jaswanth, Tushar, and Kaushik were always there to help me whenever I had a problem. Chris, Vincent, and Girish were always available when I needed help. I am especially grateful to Changhee for being by my side whenever I was struggling and for being my mentor.

Finally, I am very grateful to my family in Korea for their emotional support and to numerous other kind people I met during my years at Georgia Tech, especially my roommate, Seungwoo Jung. Although My Ph.D. study was an extremely stressful experience, my colleagues, family, and friends have provided me with unforgettable memories of this time.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Memory bloat is loosely defined as an excessive memory usage by an application during its execution. Due to the complexity of efficient memory management that developers have to deal with, memory bloat is pervasive and is often neglected in favor of lower application development cost. Unfortunately, when the bloat becomes severe, unwanted performance issues may occur due to its impact on memory management mechanisms and memory layout.

In light of this, this dissertation identifies three pervasive causes of performance issues due to memory bloat and presents feedback-driven solutions for each.

First, in certain languages like C/C++, applications have to manually manage memory in terms of allocation and deallocation. When users forget to free an allocated memory (due to a bug), this leads to a form of memory bloat known as memory leak. The presence of memory leaks causes gradual exhaustion of system memory and eventually leads to serious performance degradation of production systems. To prevent the obvious consequences of memory leaks, we present a memory leak detection framework that relies on object behavior introspection. Our framework models behavioral changes of hypothetically leaked objects in terms of their staleness and coexistence patterns among the allocated objects. With the introspective memory leak detection framework, we observed significant memory bloat savings upon weeding out the discovered memory leaks.

Second, memory bloat prevention mechanisms in multi-threaded memory allocators is another source of performance issues. When the bloat prevention mechanism is frequently triggered unnecessarily as an artifact of intensive memory allocations/deallocations, an application may experience suboptimal performance. To address

this, we present a feedback-directed tuning mechanism for TCMalloc, a widely used memory allocator for high performance systems. Our optimization technique tunes the thread cache management mechanism in TCMalloc to the memory allocation behavior of an application and reduces the management cost of the internal data structures in TCMalloc. With the proposed technique integrated into FDO in GCC, we observed up to 10% improvement in application performance.

Third, in some languages like Java, memory is automatically managed through garbage collection. Memory bloat in Java applications occurs due to performance unconscious designs and implementations. When an application uses an excessive amount of memory by creating more objects than are necessary, negative performance impacts such as high garbage collection overhead may arise. To address this issue, we present an object recycle optimization technique for Java applications. Our technique uses a static analysis to figure out safe-deallocation sites of objects and uses a dynamic profiling to select allocation sites for code transformation. With the optimization technique, We observed up to 10% improvement in application performance on the DaCapo 2006 benchmark applications.

In summary, this dissertation comprehensively analyzes and proposes solutions to the performance issues due to memory bloat in both manual and automated memory managed systems.

CHAPTER I

INTRODUCTION

Myhrvold's premise that "software is a gas" describes the tendency of software systems toward ever-growing complexity [49]. According to Myhrvold, software is constantly growing in size and complexity to fit the container it is stored in, the hardware, which scales with Moore's law [54]. Unfortunately, with complexity comes the inevitable danger of inefficiency [7].

One characteristic of large and complex software systems is their intensive use of dynamic memory [28, 8]. For applications written in unmanaged languages such as C and C++, the management of allocated memory is handled through calls to the memory allocator by explicitly marking the birth and the death points of allocated objects during the development. On the other hand, for applications written in managed languages such as Java, the detection and reclamation of dead objects are carried out automatically by garbage collection during the runtime. Whichever mechanism is used, however, the programmer is supposed to carefully manage/utilize the allocated objects. The failure to do so leads to the inefficiency known as memory bloat [47, 48, 50, 55, 67].

Memory bloat is loosely defined as an excessive memory usage by an application during its execution. Due to the complexity of efficiently managing memory, memory bloat is pervasive and is often neglected in favor of lower application development cost [47]. Unfortunately, when the bloat becomes severe, it can cause unwanted performance issues [68]. For this very reason, dealing with memory bloat has been a concern for software developers.

However, memory bloat related performance issues are still widespread for two

reasons. First, the capacity of available system memory is constantly increasing. The abundance of memory space gives application developers an illusion that using more memory does not cost much. Due to this reason, memory bloat accumulates without being noticed. In addition, reliance on libraries and frameworks to expedite application development causes dependency on external codes that are invisible to and uncontrollable by application developers. Accounting for the impact of this complex dependency is virtually impossible for an average developer. Due to these reasons, memory bloat is difficult to diagnose and its presence in a software system often leads to reduced system performance.

The performance impact and the pervasiveness of memory bloat necessitate an investigation into the causes of memory bloat related performance problems and the development of proper solutions. In this thesis, we diagnose three predominant causes of memory bloat related performance issues and present solutions based on profile-guided optimization.

1.1 Memory Bloat Due to Memory Leaks

Memory bloat often manifests in the form of memory leaks. Since a program cannot reclaim the leaked memory region, continued memory leaks increase memory footprint gradually and eventually impact the system performance. To prevent the consequences of memory bloat due to memory leaks, the latest generation of dynamic memory leak detectors use the notion of object staleness that represents how long an object remains unaccessed [24, 15, 60, 13, 14, 50].

However, the operation of these leak detectors depends heavily on the user's expectation of object access patterns. In other words, they ask the user to provide a staleness predicate, which specifies the degree of staleness above which an object can be viewed as having leaked. This dependence on the user's knowledge and the lack of analytical reasoning for leak detection causes sub-optimal use of the staleness-based

leak detection technique.

To address this, we present the introspective memory leak detection framework in Chapter 2. The introspective memory leak detection framework augments the staleness-based memory leak detectors. The proposed solution leverages what we termed *what-if-leak staleness* and *what-if-leak allocation context coexistence* to account for the behavioral changes of heap objects that might have happened if they had been lost to an application instead. By estimating the *what-if-leak* traits during a profile run and by constructing object behavior models that use the traits, the introspective leak detection framework automates staleness-based leak detection by removing the dependency on the user for configuring the staleness predicate and, more importantly, achieves improved accuracy/utility for the staleness-based technique with a model-based prediction.

1.2 Memory Bloat Inside Multi-threaded Memory Allocators

Memory bloat also occurs in a seemingly odd but critical part of a software system, i.e., the memory allocator. The latest multi-threaded memory allocators use thread-local caches for fast allocation/deallocation of memory objects [32, 30]. To manage these caches, memory allocators perform management operations. However, when the management operation is frequently triggered unnecessarily to deal with a bloated thread-local cache, performance degradation occurs.

To address this problem, we present a feedback-directed tuning mechanism in Chapter 3 for TCMalloc, a widely used memory allocator for high performance systems. The proposed optimization technique tunes the batch sizes, a set of key parameters that determine the number of objects to be moved during the thread cache management. The proposed solution observes the behavior of TCMalloc during a profile run of an application and determines proper values for the batch sizes according to a heuristic. This optimization method reduces the management cost of

the internal data structures in TCMalloc and improves the application performance. With the proposed technique integrated into Feedback-Driven Optimization in GCC, we observed up to 10% improvement in application performance.

1.3 Memory Bloat Due to Memory Inefficient Design/Implementation

Implicit memory management in managed languages such as Java is inherently vulnerable to memory bloat. For applications written in such languages, the detection and reclamation of dead objects are performed lazily by garbage collectors when an application runs out of memory. A problem occurs when the developer blindly relies too much on garbage collection for the management of allocated objects and pays little attention to the cost of using many short-lived temporary objects. In the worst case, this may increase the garbage collection pressure and cause an application slowdown.

To address this issue, we present an object recycle optimization method in Chapter 4. The proposed optimization method uses both static and dynamic analyses to recycle identifiably dead objects. With a static shape analysis, we track the liveness of allocated objects and discover the safe-deallocation sites of the objects. We use the deallocation sites to transform the program to cache dead objects and to use the recycled objects instead of creating new ones. Unfortunately, however precise a static analysis is, it is imperfect, as Java applications are written in a way that is not necessarily favorable to the static deallocation of objects. To circumvent this, we also use profiling to trim down the set of allocation/deallocation sites to only those that are useful. That is, we limit ourselves to those allocation sites that are responsible for high garbage collection pressure and their corresponding deallocation sites that are mostly computable statically. With the object recycle optimization, we reduce the memory bloat of Java applications suffering from high garbage collection pressure. On the DaCapo 2006 benchmark applications, we observed up to 10% performance improvement.

1.4 Thesis Statement and Contributions

The proposed suite of techniques in this dissertation tackles the memory bloat problem at three different levels: as a bug (memory leak), as a memory allocation/management efficiency problem (for TCMalloc), and as an optimization problem for object recycle (using both static and dynamic analyses). The techniques support the following hypothesis.

Leveraging the observed behavior of an application during a representative profile run can enable optimization of the application to mitigate the performance impact of memory bloat on a large scale.

1.4.1 Contributions

To this end, this thesis makes the following contributions:

- We present the introspective memory leak detection framework that enables the observation of application behaviors during a profiling run and the detection of anomalies indicating a memory leak during the production of the target application.

- We present a feedback-directed optimization that tunes the management mechanism for the internal data structures in TCMalloc.

- We present an object recycle optimization for reducing the performance impact of memory bloat in Java applications. The optimization technique combines a static analysis and a dynamic profiling to select optimization targets and modifies the bytecodes to recycle provably dead objects at selected allocation sites.

CHAPTER II

INTROSPECTIVE MEMORY LEAK DETECTION

2.1 Introduction

For unmanaged programming languages, memory leaks are a common bug that undermine the quality of the program. A memory leak occurs when an application omits the deallocation of an allocated memory object that has no future use. Because memory leaks gradually exhaust available system memory, they are often the root cause of performance degradation and the sudden hang/crash failure of software systems. Moreover, memory leaks can be intentionally exploited by adversaries to launch denial-of-service attacks [62].

One notable example showing the severity of memory leakage is the Amazon web services outage [63]. In 2012, Amazon replaced a data collection server. Unfortunately, this seemingly harmless maintenance action caused an incorrect configuration of some servers, which led to memory leaks. Due to the failure of a monitoring alarm, the memory leaks went out of control eventually, and the affected servers came to a stop. Consequently, millions of users were affected by the memory leaks.

What makes memory leaks hard to detect and fix at an early stage of development is their input and environmental sensitivity. Since the number of possible execution paths is potentially infinite, covering every possible execution path and configuration is not feasible even for the extensive in-house testing. As a result, only the obvious leaks are discovered and fixed during the testing stage. The remaining leaks, therefore, are highly susceptible to the execution environment and are highly elusive. The Amazon memory leak incident is a clear demonstration of how tricky memory leaks are.

In response to the insidious nature of memory leaks, researchers have proposed low overhead dynamic memory leak detection techniques. These techniques are based on object *staleness*, which represents how long an object remains unaccessed during program execution. Staleness-based leak detection relies on an intuition that leaking objects must be stale, whereas live objects with pending uses are unlikely to remain unaccessed for a long period of time. In light of this, the leak detection method assumes that memory leaks can be effectively identified by using a proper staleness predicate, which decides whether an object has leaked based on the degree of staleness of the object.

This work is an extension of staleness-based memory leak detection in that it also leverages object staleness as an indication of memory leaks. However, this work obviates using object staleness as is and the use of user-definable staleness predicates, unlike previous research. Instead, the memory leak detection framework proposed in this work is based on introspective reasoning about how leaking objects may behave (e.g., how stale the objects would be) and on modeling the behavior for detecting memory leaks.

The key idea behind using a model-based approach is that a leaking object is discernible by observing the lifetimes of other similar objects. That is, an object can be regarded as having leaked when it shows an anomalous behavior such as a high degree of staleness that is not observed from other supposedly similar objects, i.e., objects with the same allocation context. To make matters easier, once the expected behavior of an object is known, introspecting what may happen if the object has leaked is surprisingly easy, i.e., the object becomes only more stale due to the prolonged lifetime. This leads to a key insight of this work: that memory leaks can be effectively addressed by observing/modeling the behavior of similar heap objects.

In light of this, this work presents an introspective memory leak detection framework, which is intended to be used throughout the development cycle. The proposed

framework enables the observation of heap objects during the testing phase (e.g., regression testing) and the application of this knowledge during the production phase. To this end, the leak detection framework observes the target application during testing with a memory access sampling. The framework then performs a trace simulation on the stored heap activities to reconstruct the observed heap states and to generate training examples. By using these examples, which also contain hypothetically injected leaks, the framework constructs object behavior models. The framework then uses the models during the production runs to detect anomalous behavior indicating a memory leak.

The introspective memory leak detection framework has several advantages over the previous approach relying solely on user-configured staleness predicates. First, because the new approach is a framework, it can be configured to use any memory access sampling method for estimating the degree of object staleness. Depending on the runtime requirement (e.g., a custom memory allocator is not allowed), an alternative sampling method can be used with no change to the whole framework. Also, compared to the previous approach, the model-based leak detection, by virtue of being application tailored, offers the automation of staleness predicate determination. Moreover, the new approach provides increased accuracy by incorporating additional information, i.e., allocation context and object coexistence patterns, into leak detection.

The introspective memory leak framework was first evaluated on synthetic leakage workloads, which were generated by dynamically removing 5% of deallocations during the execution of the SPEC CPU2006 benchmark suite. The results of the workloads show that the proposed framework achieves the best accuracy permitted by staleness-based leak detection and improves upon it by incorporating additional information into input features of the trained models. For 5 out of 9 applications tried, the addition of allocation context and object coexistence information achieved meaningful

accuracy improvement.

In addition to the synthetic leakage workloads, the proposed framework was also tested on two publicly available open-source programs, lighttpd-1.4.19 and bash-4.0, to demonstrate how the introspective memory leak detection framework can be applied to real world scenarios. The empirical evaluations show that the proposed framework achieves very accurate results and succeeds in correctly identifying the reported memory leaks of the tested programs.

The rest of this chapter is structured as follows: Section 2.2 presents the motivation of this work. Section 2.3 describes the introspective memory leak detection framework, and Section 2.4 evaluates the performance of the proposed framework on both synthetic leakage and on real world examples. Finally, Section 2.5 summarizes the lessons learned from the work.

2.2 Motivation

Memory leaks are distinct from any other program bugs in that they do not affect program behavior directly by influencing (or altering) the execution path. Only accumulated leaks, through a form of performance degradation, are visible to the outside world. Due to this non-interactivity with program semantics, predicting what may happen to the leaking objects is relatively easy, i.e., once memory objects become lost, they cannot be accessed any more by a program. In fact, a recent generation of dynamic memory leak detection techniques based on staleness tracking exploits this simple idea [24, 15, 60, 13, 14, 50].

Staleness-based leak detection is an intuitive view of the memory leakage problem, which attributes stale objects as the symptom of memory leaks. The leak detection method is built around the idea that leaking objects must be necessarily stale, whereas other objects that are currently in use are unlikely to be so. Leaking objects, by definition, do not get accessed after they become lost to an application and become

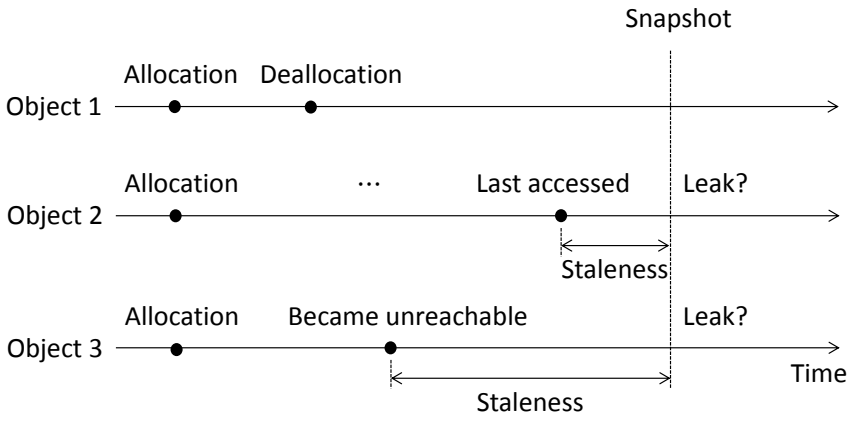

Figure 1: Staleness assumption: leaking objects are more likely to be stale than others.

stale. Objects that are still in use, on the other hand, are likely to be accessed frequently to utilize their memory space (otherwise the objects bloat the memory footprint, which wise programmers avoid through various ways).

Figure 1 is an illustration of how the staleness assumption relates to memory leak detection. Initially, three objects are allocated at a program point simultaneously. At a later time, the leak detection method takes a heap snapshot to query whether any object currently residing in memory has leaked. Object 1 is deallocated prior to the heap snapshot point, and is irrelevant to the leak query. Object 2 is a normal object that has accrued a certain degree of staleness since its last access. Object 3, unlike the previous two objects, becomes unreachable long before the snapshot point and is highly stale at the point. Using a staleness threshold that is higher than the degree of staleness of object 2 and smaller than the degree of staleness of object 3, staleness-based leak detection can successfully identify the leaking object.

In previous research, this assumption is encoded as a user-definable predicate, e.g., the user may specify that an object has leaked if it remains idle for 100 million memory accesses. However, relying on the staleness predicates for an accurate detection suffers from three shortcomings. For one thing, the appropriateness of a predicate heavily depends on the behavior of an application. Hence, the user has to find a proper

staleness predicate for every new application. Secondly, staleness estimation methods are usually based on sampling. It is unrealistic to assume that the user can account for the sampling factor in configuring a predicate. Additionally, using a user-definable predicate hampers the use of additional information such as allocation context and object coexistence patterns. For example, expressing which allocation sites can create coexisting objects [1] is not only verbose, but is often impossible to precisely specify without a tool support.

To work around this, this work takes inspiration from previous research on object lifetime prediction [6, 39]. According to these works, the lifetime of an object strongly correlates with its allocation context. Since the staleness of an object is bounded by its lifetime, object staleness is transitively correlated with allocation context. For example, it becomes more convincing that object 3 in Figure 1 has leaked if the other objects are from the same allocation context, and if they are supposed to be similar in their behavior. Similarly, due to the similarity of objects with the same allocation context, objects from an allocation context may only coexist with objects created from a limited set of allocation contexts. If the lifetime of an object becomes prolonged due to a memory leak, an anomalous coexistence pattern may be detected. With this in mind, this work assumes that an object can be considered to have leaked (with high probability), if the following two conditions are met.

1. If an object shows a significantly high degree of staleness not observed from other similar objects,

2. and if the object coexistence pattern of an object deviates from the pattern of the observed objects with the same allocation context.

To leverage this insight, the introspective memory leak detection framework proposed in this work observes the behavior of heap objects from a representative run

[1] An object coexists with another if the lifetime of the object overlaps with the other.

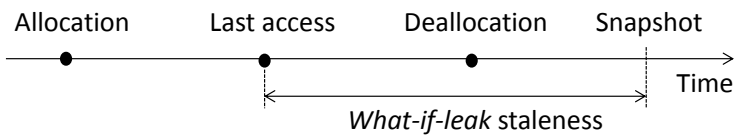

Figure 2: *What-if-leak* staleness corresponds to the time interval between the last access point of an object and the heap snapshot point.

of the target application. The observed patterns are then modeled using supervised learning to construct object behavior models, which base the decision whether an object has leaked or not on the similarity/dissimilarity of objects in terms of their lifetimes and staleness behaviors. With this approach, the proposed framework replaces the user-definable staleness predicates with a model-based prediction.

A central question that needs to be addressed in using supervised learning is how to obtain training examples. That is, a supervised learning algorithm is essentially a process of iteratively refining a model by using both positive and negative examples. Therefore, to train a model of leaking objects and live objects with pending uses, instances of both cases exhibiting their differences have to be supplied to a learning algorithm. However, getting an instance of a leaking object is, in general, extremely difficult because no a priori information is given during the testing of an application; i.e., an execution path that can trigger a potential memory leak is not yet identified at this stage (otherwise, it should have already been fixed). On the contrary, a testing environment is supposed to be a representative usage scenario of an application. Hence, the objects observable during the testing show only how normal objects will/should behave during production.

The solution comes from the characteristic of memory leaks, i.e., they do not alter the execution of a program. Suppose an object is deallocated prior to a specific time, when the heap state is examined. Since the only side effect of a memory leak is in object lifetime, the only thing that may have happened if the deallocated object had been lost instead is a prolonged lifetime of the object. This behavioral change leads

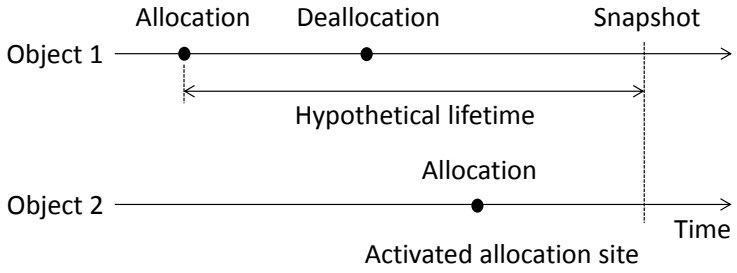

Figure 3: A prolonged lifetime of an object due to a hypothetical memory leak may result in the observation of additional memory allocations and may affect the object coexistence pattern.

to a higher degree of staleness and the observation of what was posthumous object creations that can possibly enlarge the set of coexisting allocation contexts [2]. In this work, these hypothetical traits are termed *what-if-leak* traits. Figure 2 is an example showing *what-if-leak* staleness. Assuming the object in Figure 2 had been leaked instead, the degree of staleness of the pictured object would have equaled the *what-if-leak* staleness shown in the figure. Figure 3 shows *what-if-leak* allocation context coexistence. Due to a hypothetical memory leak, the prolonged lifetime of Object 1 covers the allocation of Object 2, which is from a different allocation context. This results in enlarging the set of activated allocation sites during the lifetime of Object 1. As shown in the figures, using instances of hypothetically leaked objects enables the introspection of abnormal object behaviors with respect to memory leaks. Using these data as training examples for model construction is an approximate solution leveraged by the introspective memory leak detection framework.

The behavior models trained using the instances of hypothetically leaking objects are the models of what may happen if an object has leaked. As such, the trained models are able to handle almost all possible leakage scenarios by reflecting *what-if-leak* traits of all objects observed during testing. As long as the testing environment remains a representative of the production run, the constructed models must be

[2]We define an allocation context to be live if there is a live heap object created from the allocation context.

accurate classifiers of leaking objects.

2.3 Introspective Memory Leak Detection Framework

This section presents the introspective memory leak detection framework. To describe how the framework is intended to be used, this section discusses the overall architecture first and then continues on to the details, such as how the framework constructs object behavior models for detecting indications of memory leaks. In addition, as machine learning is leveraged inside the framework, this section discusses background information on support vector machines (SVMs) and relevant details, such as encoding object staleness as input features with respect to memory leak detection.

2.3.1 Overview

Figure 15 is an overview of the introspective memory leak detection framework. In the envisioned usage scenario, the framework is deployed throughout the life cycle of the software. When the user configures an access sampling method and runs a target application during the testing phase, the framework generates heap activity traces for an off-line analysis. During the analysis, a trace simulator reads the generated traces and replays the recorded heap activities to reconstruct the observed heap states. The heap states (or heap snapshots) are then used for generating training examples for constructing object behavior models. The training examples consist of both non-leaking objects and hypothetically leaking objects with the *what-if-leak* traits. With the training examples, the framework trains two object behavior models. The first model is called the object staleness model and corresponds to a staleness predicate for discerning suspiciously stale objects. The second model is called the allocation context coexistence model and detects an anomaly in object coexistence patterns. During the production phase, the leak detection framework applies constructed models to heap snapshots at which leak reports are requested. When both models agree that an object has likely leaked, the object is reported to the user for an inspection.

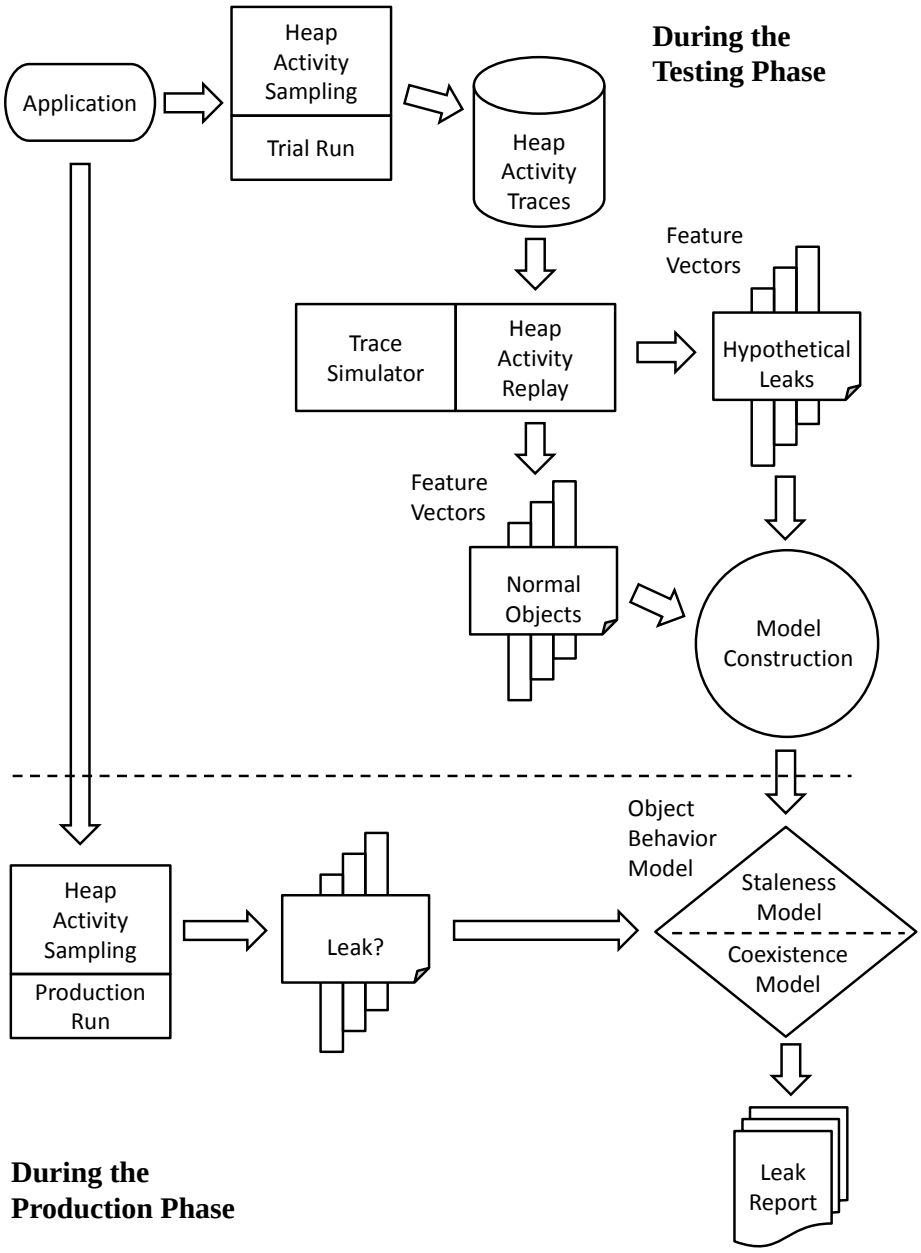

Figure 4: Overview of the introspective memory leak detection framework.

2.3.2 Trace Simulation and Training Example Generation

The introspective memory leak detection framework relies on trace simulation to generate training examples from a specific heap snapshot, which the user marks as the representative of a target application heap state. Trace simulation provides an extra flexibility by enabling a replay of the heap activities that happened during a

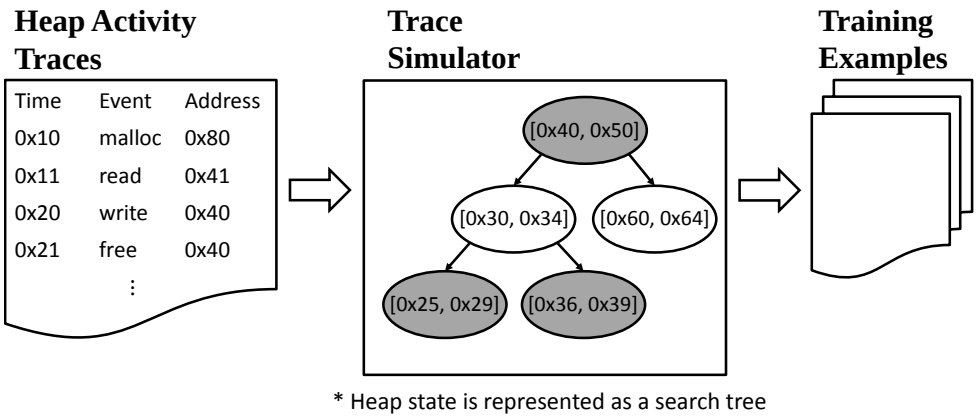

Figure 5: Trace simulation replays the sampled heap activities to obtain training examples for model construction.

profile run of an application without running it again.

Figure 5 shows how the trace simulation works. After a profile run of an application, the `malloc`/`free` traces and the sampled memory accesses are stored on the disk in chronological order. The trace simulator reads these traces and replays the operations. In doing so, the trace simulator represents the current heap state as a binary search tree, where each node corresponds to a currently live heap object. For efficient mapping of an address to the corresponding heap object, the tree data structure implements the range query with a typical binary search [3].

Each event in the traces causes an operation on the tree. A `malloc` event triggers an insertion of a new node into the tree. In addition, the allocation site of the allocated object is added to the set of activated allocation sites of each node. Similar to a `malloc` event, a `free` event triggers a deletion of the corresponding node from the tree. Each memory access forces the degree of staleness of the accessed object to 0. For other unaccessed objects, their degrees of staleness increase by the amount of time elapsed between each operation. When the trace simulator reaches a targeted heap state, training example generation starts.

[3]We implement the data structure using a specially modified RedBlack tree whose asymptotic complexities remain the same, i.e., O(logN) time [40]. This fast range search is essential for accelerating simulation.

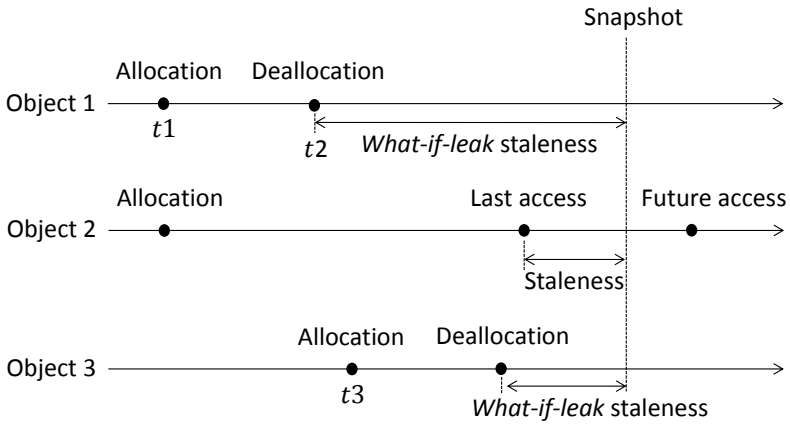

Figure 6: An example heap state reconstructed from the recorded heap activity traces. The heap snapshot in the example consists of 2 previously deallocated objects and a live object.

Figure 6 shows an example heap snapshot that the simulator reconstructs during a trace simulation. In this example, the traces contain 3 object allocation events and 2 deallocation events for 2 of the allocated objects. For an object that was deallocated prior to the snapshot point, 2 training examples are generated to account for three facts: 1) the object is supposed to be deallocated prior to the snapshot point; 2) if the object had leaked instead, the *what-if-leak* staleness is the time interval between the last access of the object and the snapshot point; 3) the deallocated object may have seen more object allocation if the object had been lost instead [4]. For an object that is still live at the snapshot, the trace simulator generates only one example to represent that an object less stale than the live object is not likely to have leaked and that the activated allocation sites during the lifetime of an object with the same allocation context are not an anomaly.

Table 1 shows the training examples generated from the heap snapshot in Figure 6. Object 1 was deallocated prior to the heap snapshot and results in 2 training examples. The first example in the first row is a negative example denoting a normal

[4]To compute the hypothetically activated allocation sites, the trace simulator sorts the allocated objects according to their allocation times and appends the set of allocation sites activated after the death of the object.

Table 1: Training examples generated by the trace simulator from the heap snapshot in Figure 6.

	Allocation Time	Degree of Staleness	Allocation Site	Object Size	Activated Allocation Sites	Label
Object 1	$t1$	0	1	8	{2}	X
Object 1	$t1$	$t4 - t2$	1	8	{2, 3}	O
Object 2	$t1$	...	2	7	{1, 2}	X
Object 3	$t3$	0	3	16	{}	X
Object 3	$t3$	...	3	16	{}	O

behavior of Object 1. The degree of staleness for this example is 0 because the object was deallocated before the snapshot point. Since the object was allocated simultaneously with Object 2, the activated allocation site column contains the corresponding allocation site. On the other hand, the second example in the second row is a positive example denoting the hypothetical behavior of Object 1, assuming it had leaked instead. Two things are notable in this example. First, the hypothetical *what-if-leak* staleness is computed as $t4 - t2$ as it is defined as the time interval between the last access point of an object and the heap snapshot point. Second, the activated allocation sites for this example contain the allocation site of Object 3 because if the object for this example had been leaked instead, it would have observed the allocation of Object 3. Object 2 is still live at the heap snapshot point. Hence, the trace simulator generates one example. For the example for Object 2, the degree of staleness is set to the current degree of staleness Object 2 has at the heap snapshot point. Example generation for Object 3 is similar to that of Object 1. However, the activated allocation sites for the positive example of Object 3 are the same as that of its negative example since Object 3 was allocated last during the profile run. Even if Object 3 had been leaked, it would not have made any difference with respect to the allocation context coexistence. With these examples, the introspective memory leak detection framework trains object behavior models.

Although the training examples are generated at a specific snapshot point, choosing the point is not a hurdle to the user. The kinds of applications that staleness-based leak detection targets are long running applications, e.g., servers or interactive games [24]. Since such applications tend to perform repetitive computations over a long period of time, taking a representative heap snapshot or specifying a snapshot point should come naturally to the user.

2.3.3 Object Staleness Model

The first model the introspective memory leak detection framework uses is the object staleness model. Basically, this model is an equivalent of the staleness predicates that is leveraged in staleness-based leak detection. However, using a model has two advantages compared to manually selecting a staleness predicate. First, this approach automates staleness predicate selection by examining a representative run of a target application. Hence, no user intervention is required other than setting up a profile run for the framework. Second, using a model enables the use of additional information (e.g., allocation context) that can improve the accuracy of leak detection.

2.3.3.1 Support Vector Machines

The introspective memory leak detection framework constructs the object staleness model using support vector machines (SVMs) [27]. SVMs are machine learning models that represent linearly separating hyperplanes. For example, when the dimensionality of an input data is 2, a SVM model corresponds to a line that divides the 2D input space into 2 mutually disjoint regions, i.e., data points that lie above the line are classified as one category, whereas data points that lie below the line are classified as the other category. Applied to memory leak detection, a SVM model is conceptually a staleness predicate, which bisects the input space representing the degrees of staleness of allocated objects into 2 regions, respectively for leaking objects and innocent objects.

The choice of SVMs among various machine learning models is due to the computational capability of the models. With more complex models, such as artificial neural networks, this work found that there is a danger of learning a contradictory function; i.e., the trained model may identify a lowly stale object as non-leaking, a modestly stale object as leaking, and a highly stale object as non-leaking.

2.3.3.2 Input Features

Machine learning algorithms require training examples to be represented as input features. For each training example, the introspective memory leak detection framework encodes the information about the object as follows.

First, instead of using the absolute degree of staleness values, the leak detection framework encodes the staleness of an object as 2 input features to normalize the input range to [0, 1]. Without normalization or scaling, input features in greater numeric ranges dominate those features in smaller numeric ranges and may jeopardize the overall accuracy [38]. The input features for an object o are as follows.

$$NT_{allocation}(o) = \frac{T_{allocation}(o)}{T_{snapshot}}$$

$$NT_{staleness}(o) = \begin{cases} 0, \text{ if } o \text{ is deallocated prior to } T_{snapshot} \\ \dfrac{T_{snapshot} - T_{last\_accessed}(o)}{T_{snapshot} - T_{allocation}(o)}, \text{ otherwise} \end{cases}$$

where $T_{allocation}(o)$ represents the allocation time of o, and $T_{last\_accessed}(o)$ represents the last access time of o [5].

Both features have the range of 0 to 1, and express relative allocation time and relative degree of staleness as percentiles of the execution time or the lifetime of an object.

[5] They are measured in units the configured staleness estimation method uses, e.g., the number of memory accesses observed.

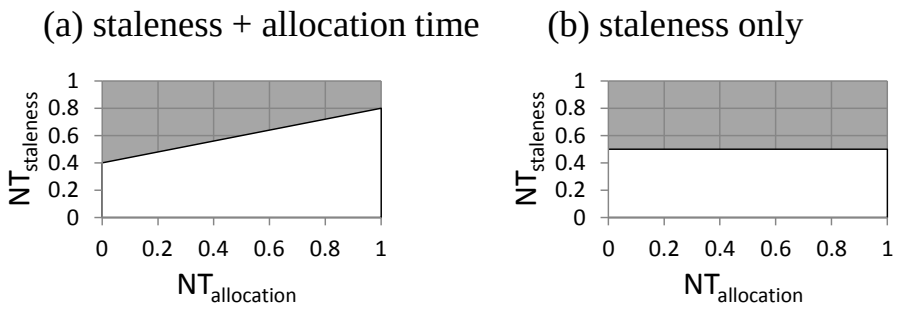

Figure 7: 2 examples of memory leak indicator functions that can be expressed using the normalized input features. If an object falls on the shaded area, it is regarded as having leaked.

Figure 7 shows 2 examples of memory leak indicator functions that can be expressed using the normalized input features. Due to the computational capability of SVMs, these functions are linear classifiers. The first leak indicator function uses the 2 input features. For this function, even a small $NT_{staleness}$ is enough as evidence for an object, which was allocated early, to be declared to have leaked. For an object that was allocated late, the indicator function is more lenient by permitting a higher degree of staleness. On the other hand, the second indicator function uses only the normalized staleness. It declares an object to have leaked if $NT_{staleness}$ is above some threshold regardless of the allocation time. These functions roughly correspond to the staleness predicates used in the previous work [24].

Using the normalized input features requires special care, however, because the features are relative to a snapshot point, i.e., $T_{snapshot}$. Since the absolute value of $T_{snapshot}$ is different for each snapshot taken during production runs, adjusting for that fact is necessary. Otherwise, if $T_{snapshot}$ of a heap snapshot is much larger than $T_{snapshot}$ of the training examples, the input features may look insignificant to the model; i.e., even if an object has the same degree of staleness and the same lifetime, the object may be seen as less stale/old if it was observed during a longer run. The purpose of using the relative input features is to normalize the ranges, not to distort the scale.

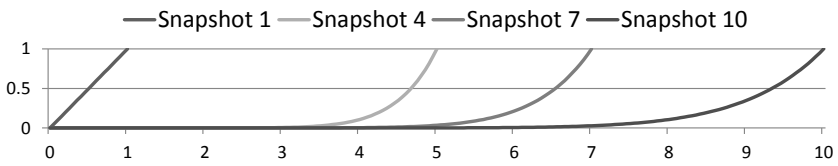

Figure 8: Scaling execution time of production runs using polynomials at heap snapshots.

To address this issue, the leak detection framework performs time scaling using a polynomial function. The scaling function maps a time of an event during a production run to a likely time when the same event might have happened during the profiling run. The polynomial function is defined as,

$$r = \frac{T_{snapshot}}{T_{train\_snapshot}}$$

$$TS(t) = T_{train\_snapshot} * (\frac{t}{T_{snapshot}})^r$$

Note that the slope of TS at $T_{snapshot}$ is 1. Therefore, for the events that happened near the heap snapshot point, the scaling factor is 1; i.e., the original scale is preserved. On the other hand, for the events that happened much earlier, the scaling factor is very small, which makes the events look as if they happened very early.

Figure 8 depicts a situation where 4 snapshots are taken during a production run. For the heap access events that happened along the x-axis, their respective times of occurrences are scaled by the respective polynomial for each snapshot. For snapshot 1, the original scale is preserved because the execution time is equal to that of the profiling run. On the other hand, moderate compaction of time occurs for snapshot 2 and 3 as the execution time becomes longer. For snapshot 10, since the execution time is 10 times longer, the time is highly compacted with some level of scale preservation near the end.

To express the similarity of objects based on their allocation context, the introspective memory leak detection framework uses the allocation site and allocation size

as additional input features. Much the same way as in encoding object staleness, the allocation size is normalized using the maximum object size observed during the profiling run.

$$NSize(o) = \frac{Size(o)}{\max_{obj \in objects} Size(obj)}$$

Encoding the allocation site, on the other hand, uses a vector of binary variables. Formally, the vector $NSite(o)$ is defined as,

$$NSite(o) = (Site_1(o), Site_2(o), ..., Site_{n-1}(o), Site_n(o))$$

$$Site_k(o) = \begin{cases} 1 & \text{if } o \text{ is allocated from allocation site } k \\ 0 & \text{otherwise} \end{cases}$$

2.3.3.3 Polynomial Kernel

Inclusion of the allocation context into the input features is based on the assumption that the context has a correlation with the estimated staleness. However, using the input features as they are does not express this relationship; i.e., the features are independent. To handle this issue, the introspective leak detection framework uses a kernel trick, which conceptually maps the input features into a higher dimension where the interaction can be expressed.

The polynomial kernel is a commonly used method in conjunction with SVMs. When input data are vectors in $\mathbb{R}^2$, the polynomial kernel with degree 2 maps an input vector to a vector in $\mathbb{R}^3$ using the following mapping [17].

$$\Phi(x) = \begin{pmatrix} x_1^2 \\ x_1 x_2 \\ x_2^2 \end{pmatrix}$$

Since $NSite(o)$ is a vector of binary variables, applying the polynomial kernel results in a conceptual feature vector, where most of the elements are zero and only the terms representing normalized staleness or the terms expressing interaction between allocation context of o and staleness of o are given non-zero values.

Using the polynomial kernel is purely for the sake of computational efficiency. Instead of using the kernel trick, the input features can be explicitly mapped into a higher dimension. Compared to explicit mapping, the kernel trick has one drawback. Terms like x_1^2 introduced by the polynomial kernel distort the shape of the indicator functions; i.e., they become curves, not lines. However, the leak detection framework prefers using the kernel trick in favor of reducing the computational load, which is expensive, even with the trick.

2.3.3.4 Cross-validation and Grid-search

SVMs have a handful of parameters that influence the accuracy of the trained model. One recommended way to determine the parameters is to use cross-validation and grid-search [38].

Given a specific set of parameters, cross-validation computes a predicted accuracy of a model trained using the parameters. In a v-fold cross-validation, the training set is first divided into v subsets of equal size. Subsequently, one subset is used as a test set while the classifier is trained on the other subsets. The process takes turns until all inputs are used as a test instance once. The averaged accuracy obtained thus is the expected accuracy of a model given a set of SVM parameters. The cross-validation procedure has a benefit of preventing the over-fitting problem, which occurs when a model fits itself to the training data too much instead of learning the general trend in the data.

To provide a set of model parameters for cross-validation, the leak detection framework uses the grid-search. The grid-search is a naive method that enumerates all the

Table 2: Allocation context coexistence model for the training examples in Table 1.

Allocation Site	Activated Allocation Sites	Pattern Changes If Leaking
1	{2}	True
2	{1, 2}	False
3	{}	False

possible sets of parameters given the ranges of individual parameters. Although there are several advanced methods that may save computation cost, the methods are not necessarily more accurate. Additionally, the grid-search has a benefit of being highly amenable to parallelization.

2.3.4 Allocation Context Coexistence Model

The second model the introspective memory leak detection framework uses is the allocation context coexistence model. This model detects anomalous object coexistence patterns by making an association for each allocation context with other allocation contexts that may be activated while the allocation context is live. Any other coexistence pattern with no corresponding association in the model is deemed an anomaly. For an object to be considered as having leaked, the leak detection framework requires this anomaly to be present unless the computed association for the allocation context of the object does not provide any useful information, i.e., the hypothetically leaking objects created from the same allocation context had exactly the same set of activated allocation contexts as the set of activated allocation contexts for the normal objects during the profiling run.

Unlike the object staleness model, model construction for the coexistence model does not involve machine learning. Figure 9 is the pseudocode of the function that constructs a coexistence model given a set of training examples. For each allocation context that appears in the training examples, the pseudocode makes two associations with other allocation contexts. First, for each object in the training examples,

```
1: function ConstructCoexistenceModel(Training Examples)
2:     coexistenceModel = φ
3:     for each example e in Training Examples do
4:         context = coexistenceModel[e.allocationContext]
5:         if e.label == Hypothetically leaked then
6:             old = context.activatedContexts
7:             context.activatedContexts = old ∪ e.activatedContexts
8:         else
9:             old = context.hypotheticallyActivatedContexts
10:            context.hypotheticallyActivatedContexts = old ∪ e.activatedContexts
11:        end if
12:    end for

13:    for each allocation context a in coexistenceModel do
14:        if a.activatedContexts == a.hypotheticallyActivatedContexts then
15:            a.patternChangesIfLeaking = False
16:        else
17:            a.patternChangesIfLeaking = True
18:        end if
19:    end for

20:    return coexistenceModel
21: end function
```

Figure 9: Pseudocode for constructing an allocation context coexistence model.

all activated allocation contexts observed during the lifetime of the object are associated with the allocation context of the object. Similarly, all hypothetically activated allocation contexts of an object also make an association with the allocation context of the object. After this step is over, the algorithm goes over all allocation contexts found in the training examples and compares the set of activated allocation contexts and the set of hypothetically activated allocation contexts of each allocation context. If an allocation context has exactly the same set of allocation contexts for both associations, then the algorithm marks this allocation context using *PatternChangesIfLeaking* flag. Table 2 is the constructed model for the training examples in Table 1.

During the memory leak detection, the introspective memory leak detection framework uses the constructed allocation context coexistence model in the following way. First, if the observed activated allocation contexts of a heap object are in accordance with (or a subset of) the set of the allocation contexts in the model, the coexistence model rules out the possibility of the object having leaked. Second, if the coexistence model has *PatternChangesIfLeaking* flag unset for the allocation context of an object, the model entrusts the decision to the object staleness model. Third, when *PatternChangesIfLeaking* flag is set, the coexistence model decides an object as having leaked only when an observed activated allocation context of the object is not included in the set of the activated allocation contexts for the allocation context of the object. In other words, the use of an allocation context coexistence model in the introspective leak detection makes the leak detection delayed for an object whose allocation context has *PatternChangesIfLeaking* flag in the coexistence model.

2.4 Evaluation

This section discusses the design and results of the experiments that were performed to test the introspective memory leak detection framework.

2.4.1 Accuracy Metrics

To measure and compare the accuracy of the proposed method, this work uses the classification accuracy as well as the precision and recall that are the standard metrics capturing the performance of an information retrieval system. In the context of memory leak detection, they are defined as follows.

$$accuracy = \frac{|correctly\ identified\ objects|}{|objects|}$$

$$precision = \frac{|reported\_leaks \cap real\_leaks|}{|reported\_leaks|}$$

$$recall = \frac{|reported\_leaks \cap real\_leaks}{|real\_leaks|}$$

Since the classification accuracy is the metric used during the cross-validation, it is the primary metric in comparing the accuracy of the introspective memory leak detection framework. The precision and recall are of special importance because the leak report is given to the user who has to examine the cause. The first priority of a memory leak detector is to provide high precision while still identifying many true instances of memory leaks.

2.4.2 Implementation

To evaluate the performance of the introspective memory leak detection framework, this work implemented adaptive burst tracing (ABT). ABT is the memory access sampling method proposed by Chilimbi and Hauswirth in their work called SWAT, which pioneered the category of staleness-based memory leak detection [24]. However, unlike the original implementation, ABT was implemented using the LLVM infrastructure [44] as an IR pass, and it was modified to write the sampled heap activities onto the disk (for both profile and production runs). The ABT implementation used the default configuration for SWAT, which was reported to have less than 5% of runtime overhead [24]. To train SVM models, this work used LIBSVM [19] with 5-fold cross-validation.

2.4.3 Synthetic Leakage

Generally, evaluating the accuracy of a staleness-based leak detection is a daunting task for two reasons. Firstly, there is no standard set of applications containing memory leaks with which the accuracy can be measured and compared. Secondly, a staleness predicate can be applied on a set of allocated objects at any time during the execution. This means that the accuracy is snapshot point dependent.

This work addresses the first problem by synthetically injecting leaks during the execution of SPEC CPU2006 benchmarks to generate leakage workloads. Among the total 19 C/C++ application in the benchmark suite, this work selected 9 applications,

based on the number of object allocations during the application execution as well as the object deallocation pattern, in order to prevent misleading results. Additionally, this work also took into consideration heap snapshot size for the ease of training and experimentation.

In generating leakage workloads, the leakage percentage was an important factor. Too much leakage tends to inflate the precision of the resulting detection. If the portion of the leaks is too large compared to the portion of innocent objects at a program point, blindly identifying an object as having leaked becomes probabilistically a correct decision; e.g., if there are 999 leaking objects and 1 innocent object, even an inaccurate leak detector, which identifies every object as having leaked, attains a very high precision.

On the other hand, workloads with too little leakage also end up with biased results. For example, if there is only 1 leaking object and 999 innocent objects, even an inaccurate leak detector, which identifies no object as having leaked, attains a high accuracy. Thus, both very high and very low percentage of leaks endanger the evaluation of a leak detection method by producing biased results. Considering such impact of the leak percentage on the accuracy of a leak detection method, we decided to inject leaks by randomly removing 5% of the total deallocations for each application.

To address the second problem, this work fixed 95% of the execution as the snapshot point to account for the fact that the applications eventually terminate. Unlike long-running applications such as server applications that are the targets of staleness-based leak detection, the benchmark applications pass through several program phases, which may also interact with model training. Using a model that trained on examples gathered during a specific program phase may be inadequate to detect leaks during a different program phase. In consideration of this, this work gathered the heap snapshots from 95% of the execution points from the runs of the applications

Table 3: Trace file (in text format) sizes for SPEC CPU2006 applications.

Application	Heap Activity Trace Size (in GB)	
	Train Input	Reference Input
milc	0.37	12
gobmk	0.22	2.2
povray	0.24	6.9
h264ref	5	25
astar	0.99	5.9
gcc	0.03	0.42
soplex	0.07	3.9
hmmer	2.9	20
libquantum	0.08	17

on the *train* inputs, and applied the trained models on the corresponding execution point from the runs of the applications on the *reference* inputs.

2.4.3.1 Simulation Overhead

One concern for using the introspective memory leak detection framework is its simulation overhead. Even though the leak detection framework performs most of the analysis offline, huge trace files and long simulation times may make the leak detection method less useful in reality. Fortunately, even without a clever technique to address these issues, the results on synthetic leakage workloads show negligible space and time overheads.

Table 3 shows trace file sizes for SPEC CPU2006 applications. Trace file size depends on the number of object allocations as well as the number of memory accesses during a program execution. For most applications, the trace file size on the *train* input is less than 1 GB even in text form. Had the files been in binary format, they would have consumed less space. The trace files on the *reference* inputs are bigger than their counterparts since the *reference* inputs take a longer time to process. These trace files are redundant in that they contain all memory allocations and deallocations that are not relevant to memory leak detection. The memory allocations and memory accesses that have a corresponding deallocation in a trace file serve no purpose as the

Table 4: Simulation times for generating training examples on the SPEC2006 applications.

Application	Simulation Time (in seconds)
milc	9.44
gobmk	6.22
povray	40.52
h264ref	140.22
astar	357.43
gcc	29.07
soplex	2.26
hmmer	72.08
libquantum	2.09

object is not even live for a current heap snapshot. Hence, the redundant trace entries do not need to remain in a trace file. Because of this, trace files can be processed to reduce the size and, accordingly, to minimize the impact on the usefulness of the introspective memory leak detection framework.

Table 4 shows simulation times for the SPEC2006 applications. The trace simulator was run on the traces generated from the runs of the applications on the *train* inputs. For most applications, the simulation time was less than 5 minutes. Only *astar* took more than 5 minutes due to the frequent memory accesses and the complex allocation context coexistence pattern during the execution.

2.4.3.2 Accuracy Results

To see how well the introspective memory leak detection framework functions, this work additionally conducted experiments on the synthetic leakage workloads to obtain the optimal accuracy achievable by using the previously used approach; i.e., we experimented with several manually selected staleness predicates and picked the highest accuracy attained. In addition, to evaluate the effectiveness of using machine learning for constructing an object staleness model, this work constructed baseline models that use only the normalized allocation time and the normalized degree of

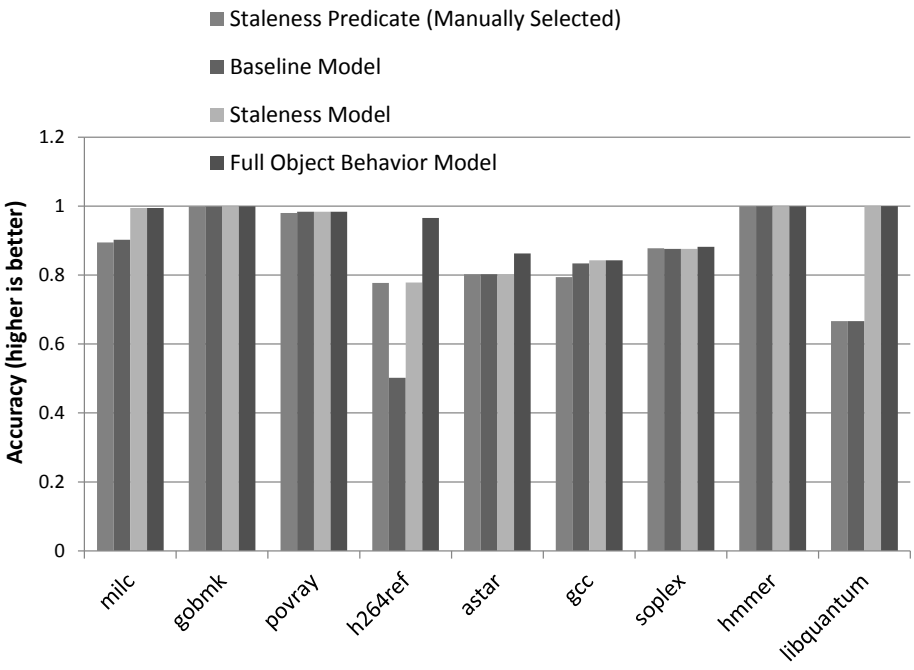

Figure 10: The accuracy of the introspective memory leak detection framework on the synthetic leakage workloads.

staleness. If machine learning succeeds in training good baseline models, the accuracies of the constructed models must be comparable to those of the manually selected staleness predicates.

In contrast to the baseline model, the object staleness model in the introspective memory leak detection framework uses allocation context information. This work evaluated the constructed object staleness models in isolation to measure the impact of incorporating allocation context information into leak detection. Lastly, the full object behavior model that combines the results of both an object staleness model and an allocation context coexistence model is what the introspective leak detection framework uses. The results on the full models in comparison to the object staleness models show the impact of leveraging an object coexistence pattern.

Figure 10 is the resulting accuracy of using the introspective memory leak detection method. It shows the classification accuracies of the constructed models. Except for *h264ref*, the accuracy of the baseline models matches that of the manually selected

staleness predicates. This result is in accordance with the expectation that the accuracy of a machine learning model is only as good as the predictive power of the input features of the model and that machine learning does succeed in training good models. A close inspection of the exceptional result on *h264ref* revealed that the profile run on *train* input was not representative of the production run on *reference* input; i.e., even the best staleness predicates manually specifiable were different for the 2 runs. This resulted in a biased model, and impacted the leak detection accuracy.

Compared to the baseline models, the object staleness models that leverage allocation context information achieved better results on 4 applications, i.e., *milc*, *h264ref*, *gcc*, *libquantum*. As expected, leveraging additional information improved the leak detection accuracy. Moreover, the full object behavior models that combine both an object staleness model and an allocation context coexistence model achieved even better results for *h264ref* and *astar*. This result is evidence that an object coexistence pattern is a meaningful indicator of a memory leak, making the introspective leak detection more accurate.

Figure 11 shows the resulting precision and recall of the introspective memory leak detection framework. Notably, the recalls are almost perfect for the constructed models. However, due to the staleness over-approximation of ABT as discussed in [24, 50], the precision is limited for *h264ref*, *astar*, *gcc* and *soplex*. The objects allocated in these applications tend to have relatively long lifetimes compared to the other applications. Such objects impacted the accuracy. Compared to the baseline models, the full object behavior models achieved better precision by incorporating the allocation context and object coexistence pattern as shown in *milc*, *h264ref*, and *libquantum*. The drastic precision gain in *libquantum* (about 50%) is because the heap snapshot contained very few leaking objects and the baseline model reported a few additional false positives. Another impressive precision gain in *h264ref* (about 40%) attributes to 2 characteristics of the coexistence model. For one thing, the measurement of the

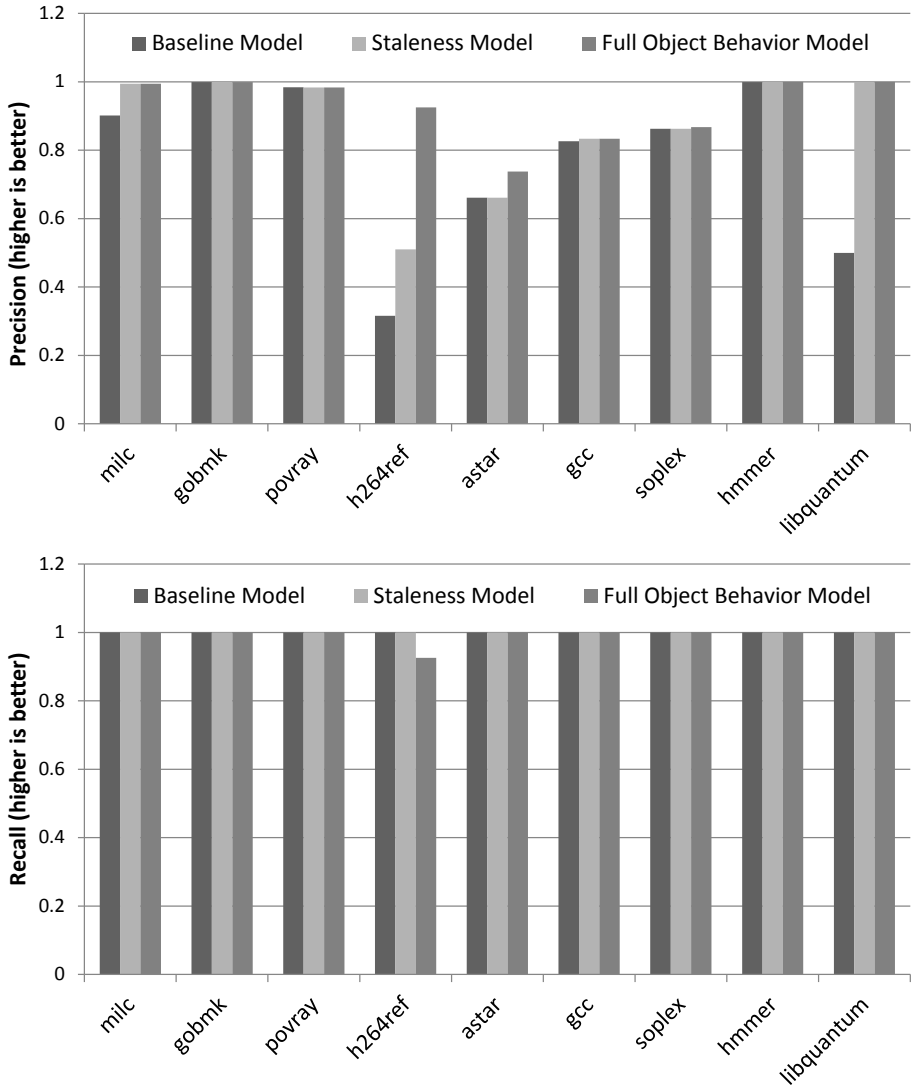

Figure 11: The precision and recall of the introspective memory leak detection framework on the synthetic leakage workloads.

object coexistence pattern leveraged in the coexistence model is independent of the memory access sampling in ABT, unlike the measurement of the degree of object staleness. For another, the object coexistence pattern is scale-invariant, i.e., it remains the same regardless of execution time variation. These characteristics of the object coexistence pattern were most significant in *h264ref*.

2.4.4 Case Studies

To demonstrate how the introspective memory leak detection technique applies to real world leakage detection problems, we tested the technique on 2 open source applications, *lighttpd* and *bash*. These applications have reported memory leaks that can be triggered only through a certain configuration/input. We compiled the applications statically to obtain the allocation context information for their libraries [6]. We also used synthesized inputs for both applications. The inputs for model construction exercise the relevant functionalities of the applications but do not introduce memory leaks. However, the inputs for testing the proposed technique periodically trigger execution paths that cause memory leaks. The evaluation results show the performance of the full object behavior model [7].

2.4.4.1 lighttpd-1.4.19

lighttpd is a popular web server that is optimized for low memory footprint [2]. In version 1.4.19, *lighttpd* has a memory leak that can be triggered by a tricky configuration using *mod_rewrite* [3].

Figure 12 is the source code related to the memory leak. At line 8, *mod_rewrite* creates a memory object that stores the current context information. A reference to the temporary context object is stored at line 9. However, this code region can be visited multiple times during a processing of a request if *url.rewrite-repeat* rule intervenes. When visited again, *mod_rewrite* creates another context object that is redundant at line 8, and overwrites the reference to the old context object by a reference to the newly created one. Since the reference to the old context object is lost by the rewriting, this results in a memory leak.

One plausible explanation for why the developers failed to see the leak-inducing

[6] The alternative is to instrument the dynamic loader (ld.so) so that it can leave the information in which the libraries are loaded [41].

[7] All features described in section 2.3 were used for constructing an object staleness model.

```
1 for (i = 0; i < p->conf.rewrite->used; i++) {
2     ...
3
4     if (...) {
5     } else {
6         ...
7         pcre_free(list);
8         hctx = handler_ctx_init();
9         con->plugin_ctx[p->id] = hctx;
10        if (rule->once)
11            hctx->state = REWRITE_STATE_FINISHED;
12
13        return HANDLER_COMEBACK;
14    }
15 }
```

Figure 12: The relevant parts of *mod_rewrite.c* in *lighttpd* that can cause a memory leak.

execution path is because *mod_rewrite* is a plug-in that is usually executed only once, per a service request. So, the developers may have assumed that the lifetime of a handler context object is finished after processing a *mod_rewrite* rule, and may have also assumed that the object will be deallocated with a connection object when the connection is reset.

To catch the leaks, we ran *lighttpd* for about an hour and let it process random service requests during the time frame for the generation of training examples. Among the requests, we included URLs that would be processed by *mod_rewrite*. However, no leak-inducing URLs were given to *lighttpd*. So, the heap activities observed contained only the expected normal operations. On the other hand, we triggered the leak-inducing execution path during the testing run. Only 1 out of every 10,000 requests contained a harmful URL.

Figure 14 is the accuracy achieved by the introspective memory leak detection framework. The accuracy measures are the averages (geometric means) of the metrics at 5 heap snapshots that were taken after *lighttpd* had been run for a long enough

amount of time, i.e., longer than the profile run. The precision is above 90% and almost perfect recall was achieved (without using the time scaling, the precision drops by about 16%). Obviously, the introspective memory leak detection framework captured the developer's assumption by learning that the lifetime of a handler context must be short and should not span multiple service requests. Grouping the reported objects by allocation context and ranking the allocation contexts revealed the problematic allocation site (line 2) as the potential cause of the mismatch with the trained models.

2.4.4.2 Bash-4.0

Bash-4.0 has a memory leak in the *read* built-in when the number of fields read from an input is not the same as the number of variables passed as arguments to the built-in [1].

Figure 13 is the relevant part of the source code that binds the last variable with the remaining input string. At line 5, a temporary memory object is created inside *get_word_from _string*, and *input_string* is advanced to the next character in the original input string to check for the end. If the next character is null, the reference to the temporary object is stored in *tofree* (line 8), and the object gets deallocated at line 27. However, if the next word is not null (line 9), the temporary object t gets lost.

To model the behaviors of objects created at line 5, we ran a *Bash* script that exercises the *read* built-in by taking synthesized inputs. The inputs for training were all well-formed and had a matching number of fields as the number of *read* variables. Thus, no memory leak was observed during the profile run. During the testing run, we supplied different inputs to a *Bash* script. The inputs were crafted so that 5% of them contain more fields than the number of *read* variables to trigger the execution path that causes a memory leak.

```
1 tofree = NULL;
2 if (*input_string)
3 {
4     t1 = input_string;
5     t = get_word_from_string (
6       &input_string, ...);
7     if (*input_string == 0)
8         tofree = input_string = t;
9     else {
10        input_string =
11        strip_trailing_ifs_whitespace (
12          t1, ...);
13    }
14 }
15
16 if (saw_escape)
17 {
18    t = dequote_string (
19      input_string);
20    var = bind_read_variable (...);
21    xfree (t);
22 }
23 else
24    var = bind_read_variable (
25          ..., input_string);
26 ...
27 FREE (tofree);
```

Figure 13: The relevant parts of *read.def* in *Bash* that can cause a memory leak.

Figure 14 is the accuracy of the introspective memory leak detection. The achieved precision is 93%, and the recall is 88% (without using the time scaling, the recall drops by about 2%). Although the detection accuracy is high enough, because the allocation site of the leaked object is inside *get_word_from_string*, tracking down the cause involved more labor. Manual inspection of the source code revealed that among the 2 callers of *get_word_from_string*, only the part listed in figure 13 has a potential memory leak.

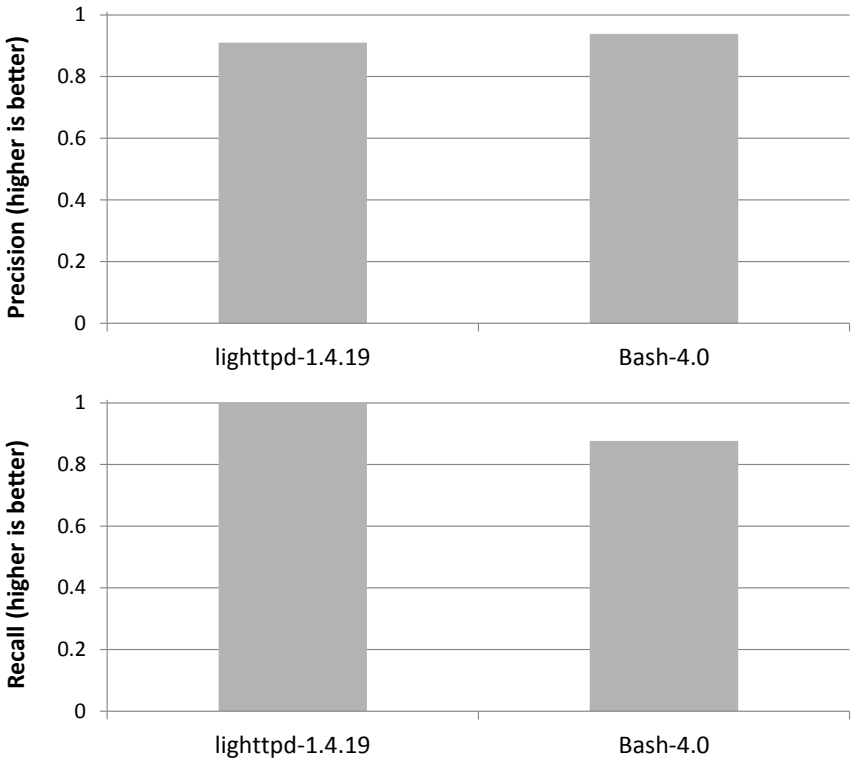

Figure 14: The accuracy of the introspective memory leak detection on 2 real-world examples.

2.4.5 Limitations

One limiting factor of the introspective memory leak detection framework is the training time, as it leverages machine learning algorithms for construct object staleness models. During the experiments, the longest training time took several days.

There are several unexplored solutions to this. The first one is to parallelize the learning algorithms [18, 34]. Sampling training examples to reduce the size of the training data is also a viable option. Lastly, instead of the naive grid-search, a guided search through the space of SVM parameters may be used to reduce the training time.

2.5 Summary

As seen from the Amazon example, memory leaks are still a major source of program instability and service interruption. Unfortunately, detecting memory leaks remains a difficult problem, as memory leaks take a long time to manifest and often do so non-deterministically based on a racing event.

To address the issue of detecting memory leaks, this paper presents the introspective memory leak detection framework that can be used with previously developed staleness estimation methods. The proposed framework is built around an idea that an object can be considered to have leaked if the object shows an anomalous behavior, such as a high degree of staleness that is not observed from other similar objects. To leverage this insight, the leak detection framework is intended to be deployed throughout the development cycle. From a representative run of an application, the framework observes the behaviors of allocated heap objects and constructs object behavior models. The object behavior models learn/introspect potential behavior changes of objects with respect to the occurrences of memory leaks. With the constructed models, the introspective memory leak detection framework replaces the staleness predicates used in previous research with a model-based detection.

The model-based memory leak detection has two obvious benefits over the previous approach. First, as the trained models reflect the observed behaviors of heap objects, the model-based approach is tailored for each target application. As such, the model-based leak detection removes the error-prone user intervention and achieves the best possible results permitted by the staleness-based approach. Second, the model-based leak detection also improves the detection accuracy by leveraging more data, i.e., the allocation context and object coexistence pattern.

Empirical results verify the usefulness of the introspective memory leak detection method. Evaluation on synthetic leakage workloads, which were generated by

dynamically removing deallocations during the execution of SPEC CPU2006 benchmark suite, shows that the introspective leak detection method achieves better results on 5 out of 9 applications than the previous approach relying solely on user-definable staleness predicates. In addition, evaluation on real-world applications demonstrates that the leak detection method succeeds in correctly identifying the reported memory leaks of the tested programs.

CHAPTER III

TUNING MEMORY BLOAT PREVENTION MECHANISM IN TCMALLOC

3.1 Introduction

Scalability of the memory allocator is a critical factor in the high performance of large-scale multi-threaded applications. In applications such as web servers and database managers, a large number of threads make intensive use of dynamic memory allocation. Hence, lock contention during memory allocation has a negative impact on the performance of the applications. This scalability issue is especially apparent when a single heap allocator is used for the applications, since the memory allocator serializes all memory allocations and deallocations with a single lock [43, 61]. Unfortunately, the scalability issue persists even for multiple heap allocators, as the alleviated lock contention is insufficient for highly multi-threaded applications [31]. In response to this, many current memory allocators take a design approach that makes a large fraction of memory allocations/deallocations lock free [46, 33, 4, 30, 32].

TCMalloc [32] is one such memory allocator that is used in large-scale multi-threaded applications. TCMalloc provides high scalability for applications through thread-local caches of free objects. When an application thread requests a new memory object, TCMalloc returns one from the local cache of the requesting thread. Likewise, when an application thread returns a used object, TCMalloc inserts the object into the cache of the thread that executed deallocation. As long as the allocation or deallocation is completely satisfied by the thread-local cache, which is true for the vast majority of requests in practice, no locking is required. Hence, TCMalloc does not require synchronization during most memory allocations/deallocations.

A major complexity of TCMalloc is in the way it maintains the thread caches. The thread cache management in TCMalloc has two conflicting goals. First, the memory allocator tries to prepare for future allocations by maintaining free objects in the thread caches. Thus, TCMalloc prefetches multiple objects from the central heap when a thread cache is empty. At the same time, TCMalloc tries to reduce the size of the caches of threads with low allocation activity. To this end, TCMalloc implements two garbage collection mechanisms called scavenge and list truncation, respectively to detect a bloated thread cache and to flush excessive free objects back to the central heap.

For large-scale multi-threaded applications, the thread cache management in TCMalloc has a performance implication for two reasons. First, the operations involving the central heap require holding necessary locks. Hence, a poorly managed thread cache could result in frequent locking, thereby limiting application scalability. Second, TCMalloc accesses its allocation metadata when it operates on the central heap, which can displace application data from data caches and TLBs, resulting in longer memory latency. Consequently, efficient thread cache management is crucial for obtaining optimal performance of applications that use TCMalloc.

One set of parameters that are critically important to the thread cache management in TCMalloc are the batch sizes. The batch sizes are defined for each predefined allocation size (named *size class* in TCMalloc). The batch size for a *size class* can be thought of as the quantum of objects of that size class that gets transferred between the central heap and thread caches. The batch sizes influence the aggressiveness of prefetching, and therefore the timing and frequency of garbage collections. As a result, the batch sizes control the access to the central heap and have a significant impact on the overhead of the thread cache management. Applications with many threads are particularly sensitive to these effects, as the total thread cache size across all threads is limited. Therefore, as the number of threads is increased, the size of

each thread cache is reduced, and it becomes even more important to balance the amount of prefetching for each size class with potential thread cache bloat.

Currently, TCMalloc uses a single static batch size for most size classes across all applications. However, the optimal values for the batch sizes for each class depend upon application behavior, specifically, the application's allocation and deallocation patterns.

In light of this, this paper presents a feedback-directed optimization of the thread cache management in TCMalloc, specifically targeting the batch sizes. The proposed optimization makes the prefetching aggressive by increasing the batch sizes for each class independently. At the same time, the optimization tries to ensure that the overhead of garbage collection is kept minimal. To achieve these goals, the proposed optimization method gathers various TCMalloc statistics during a profile run. The statistics are then used for estimating application specific thread cache budget and for determining proper batch sizes for each size class through an iterative budget allocation.

This work evaluates the proposed optimization method using two synthetic and two large benchmark applications used internally at Google. Experiments on the synthetic benchmarks show that the proposed method achieves its targeted goals. The experiments on the internal benchmarks show that the proposed method results in performance gains on large multi-threaded applications.

The rest of this chapter is organized as follows. Section 3.2 reviews key aspects of TCMalloc and discusses a performance problem that can arise due to excessive prefetching. Section 3.3 presents the feedback directed optimization. Section 3.4 evaluates the proposed solution using two synthetic and two Google internal benchmark applications. Finally, Section 3.5 concludes with the findings.

3.2 TCMalloc: Thread-Caching Malloc

TCMalloc is a memory allocator optimized for large-scale multi-threaded applications. This memory allocator performs most allocations and deallocations with no synchronization overhead by satisfying these requests from thread-local caches that are accessed lock-free. This section provides an overview of TCMalloc and discusses a performance bottleneck that arises from sub-optimal thread cache management.

3.2.1 Overview

Figure 15 shows the flow of memory objects and relevant internal data structures used inside TCMalloc during allocations/deallocations.

The heap is centrally-managed by the central page heap and central cache across all threads in the application. All accesses to the central heap therefore require obtaining a lock to enforce synchronization. Additionally, each thread is assigned a thread-local cache that does not require locking. Large objects (over 32K at least) are allocated directly out of the central page heap. Smaller objects are mapped onto one of at least 78 size classes for different allocation sizes. Allocations of small objects are satisfied by the thread caches and central cache. Both the thread caches and the central cache contain lists of free objects for each size class. The remainder of this paper discusses small object allocation.

Memory allocation in TCMalloc starts by querying a thread cache for a free object of the requested size class. If available, memory allocation is served from the thread cache. Otherwise, TCMalloc reaches for the central free list to construct a list of free objects and to fetch the list to the thread cache. When even the central free list has no free object, TCMalloc fetches a *span*, an internal data structure for a block of contiguous pages. The fetched *span* is sliced as an array of objects for the requested size class at the central free list. Memory allocation can then continue using the sliced objects.

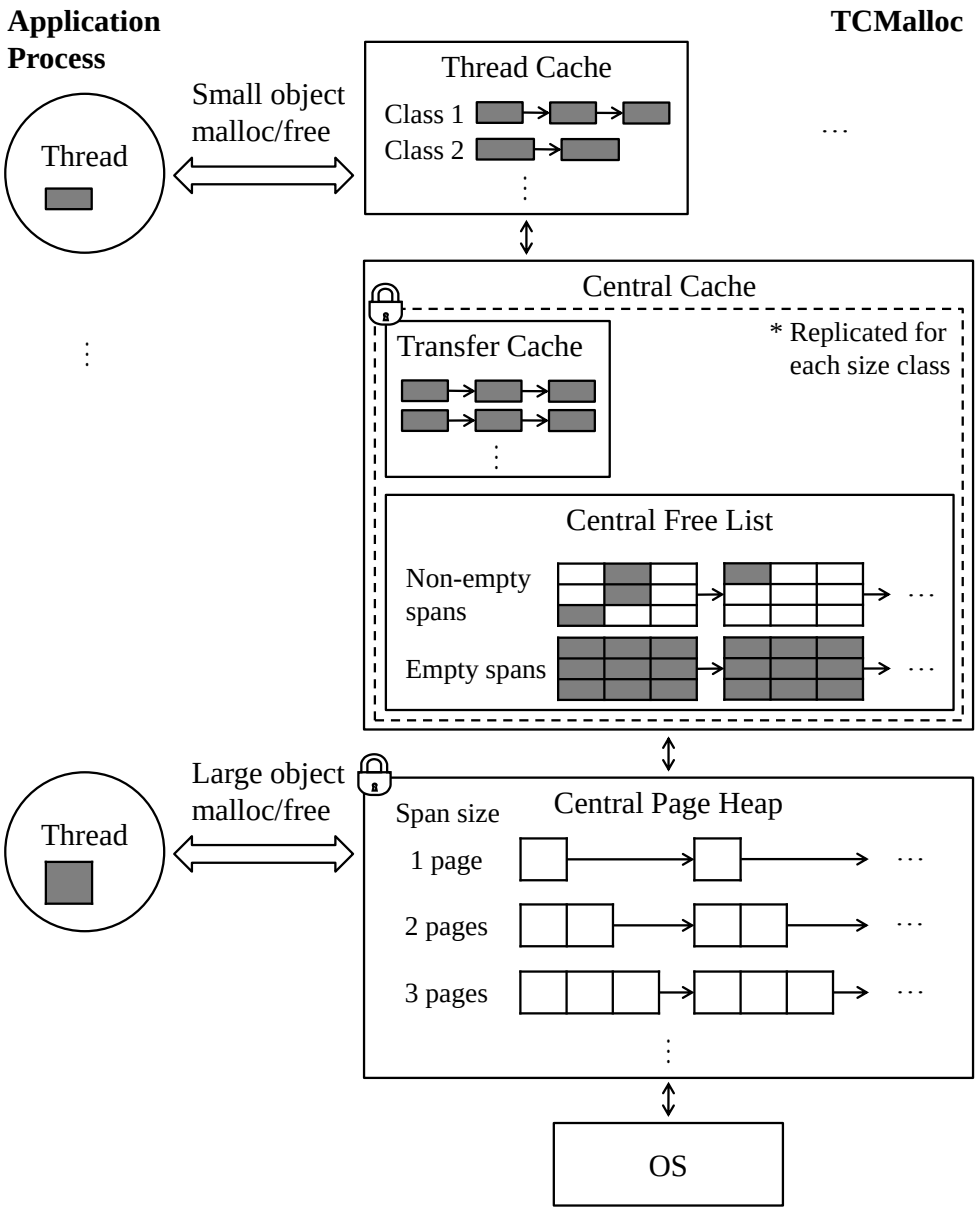

Figure 15: Memory allocation/deallocation in TCMalloc.

Deallocation works in the opposite direction. When an application thread is done with an allocated memory object, the object is added to the local thread cache. TCMalloc detects when a thread cache becomes too big and garbage collects excessive free memory objects. When no part of a *span* is being used by an application, i.e., the *span* is fully available, TCMalloc flushes the *span* back to the central page heap where it may be reclaimed by the OS.

Like many other memory allocators, TCMalloc avoids using individual object headers. Instead, the memory allocator uses *span* as the internal data structure for storing allocation metadata. Because of this, each memory object in a *span* has exactly the same size, which corresponds to one of the size classes. Due in part to this design decision, the central cache structures are also defined per size class.

Among the TCMalloc operations, the ones involving the central cache and the central page heap are costly, as they require holding a lock. In particular, creating a new list of free objects mandates crawling non-empty *span*s in search of available free objects. Doing so may increase synchronization overhead by holding a lock for too long. TCMalloc relieves this by using a transfer cache where previously constructed lists of free objects are stored. These lists must be equal in length to the batch size requested by the thread caches.

3.2.2 Thread Cache Management

The complexity and ingenuity of TCMalloc is in the way that it maintains the thread caches. TCMalloc controls the thread caches using prefetching from the central free list, as well as two garbage collection methods on the thread cache free lists called list truncation and scavenge.

3.2.2.1 Prefetching

TCMalloc fetches free objects from the central free list only if the thread cache is empty during an allocation. In doing so, the memory allocator brings in multiple objects in a single batch to prepare for future allocations. We use the term *batch sizes* to refer to the parameters that determine how many objects are fetched during this operation. The batch sizes are statically configured for each size class.

3.2.2.2 List Truncation

A thread cache in TCMalloc is a collection of linked lists of free objects for each size class. One way TCMalloc controls the free object lists is by limiting the growth of each list with a max length. When a list reaches its max length during a deallocation, TCMalloc performs an operation called list truncation, which flushes a batch of objects, where the size of the batch is the statically configured count.

The max length is dynamically adjusted during the runtime. Initially, TCMalloc sets max lengths of all lists to 1. As an application thread makes use of a particular size class, the max length of the list for the size class goes through a slow-start, which monotonically increases the length by 1. Eventually, the max length becomes the batch size of the size class. Thereafter, TCMalloc adjusts the max length by the batch size as it detects the need to increase/decrease the length.

Forcing the max lengths of free object lists to be a multiple of their batch sizes and truncating the thread caches in units of the batch sizes have one major implication: i.e., objects flushed through list truncation are saved in the transfer cache for future fast allocation back into a thread cache. For this reason, list truncation is a preferable way of garbage collecting a thread cache.

3.2.2.3 Scavenge

Unlike list truncation, the purpose of scavenge is in limiting the overall size of thread caches across all size classes. To achieve this, scavenge re-balances the capacities of multiple thread caches simultaneously while it garbage collects the thread cache that hits a capacity limit. Each scavenge operation is performed during deallocation by the thread that triggered it.

Figure 16 shows how scavenge operates. During a deallocation in thread 1, TCMalloc detects that the capacity limit of the cache for thread 1 is reached. The capacity limit controls how many free objects can be held in that particular thread cache.

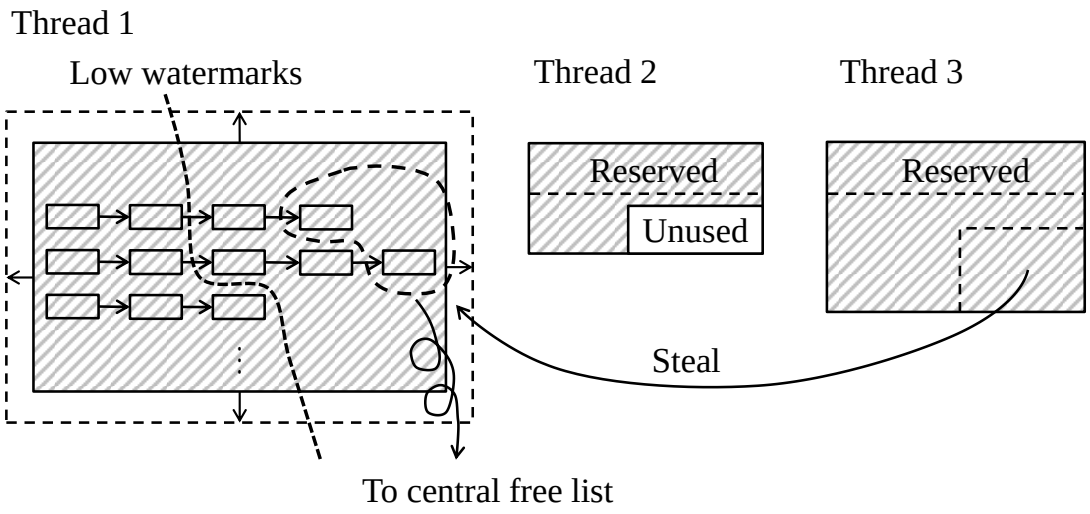

Figure 16: Thread cache capacity adjustment during a scavenge. TCMalloc flushes excessive free objects and steals cache space from another thread cache.

To make available free space for future deallocations in the thread cache, TCMalloc first flushes the free objects currently residing in the thread's own cache across all size classes. When possible, scavenge also "steals" thread cache space from another thread to account for potential asymmetry in allocation/deallocation behavior [1]; i.e., thread 1 may be the only thread performing allocation/deallocation. Scavenging requires the thread to hold a lock.

The number of free objects flushed during a scavenge is determined by the low watermarks observed since the previous scavenge. The watermarks estimate how many free objects were excessive between two scavenges. Initially, after a scavenge, the low watermarks are set to the number of remaining free objects for each size class. When the number of free objects for a size class goes below a watermark before the next scavenge, the watermark is reset to the current level. So, if a watermark is L, it means all previous allocations at that size class could have been satisfied from the thread cache without L additional free objects in the cache. TCMalloc leverages this fact and flushes half the number of free objects the watermark specifies at each

[1] Currently, TCMalloc does not prioritize a thread cache with unused space in stealing the cache space.

scavenge.

However, watermarking has one drawback. The objects garbage collected through this procedure are unlikely to hit the transfer cache, as the number of objects is not guaranteed to be equal to the batch size. This makes scavenge less preferable than list truncation.

3.2.3 Thread Cache Bloat Due to Excessive Prefetching

A problem in the thread cache management arises from ignoring the availability of thread cache space during the prefetching. Inherently, the prefetching and the garbage collection methods work in opposite directions. Therefore, the operations have to be carefully coordinated so that they do not increase the pressure for each other. Otherwise, objects can unnecessarily move back and forth between thread caches and the central free list.

Figure 17 depicts a scenario where this problem happens. In this scenario, an application thread allocates/deallocates an object in class j and four objects in class i. Suppose the configuration for the batch sizes and the thread cache capacity at the moment is as shown on the figure. Initially, the thread cache contains four free objects only for class i. As the thread cache is empty for class j, TCMalloc performs a prefetch during the first allocation. Not knowing that the thread cache capacity is five times the size of class j, the memory allocator accidentally bloats the thread cache by fetching four objects. As a result, deallocation of the allocated object is immediately followed by a scavenge, which flushes two objects in class i. Subsequent allocations of four objects in class i go through a similar path. Since the remaining two objects in the cache are not enough to fulfill four allocations, TCMalloc performs one more prefetch. After two deallocations of class i, the thread cache surpasses the capacity limit, and another scavenge occurs.

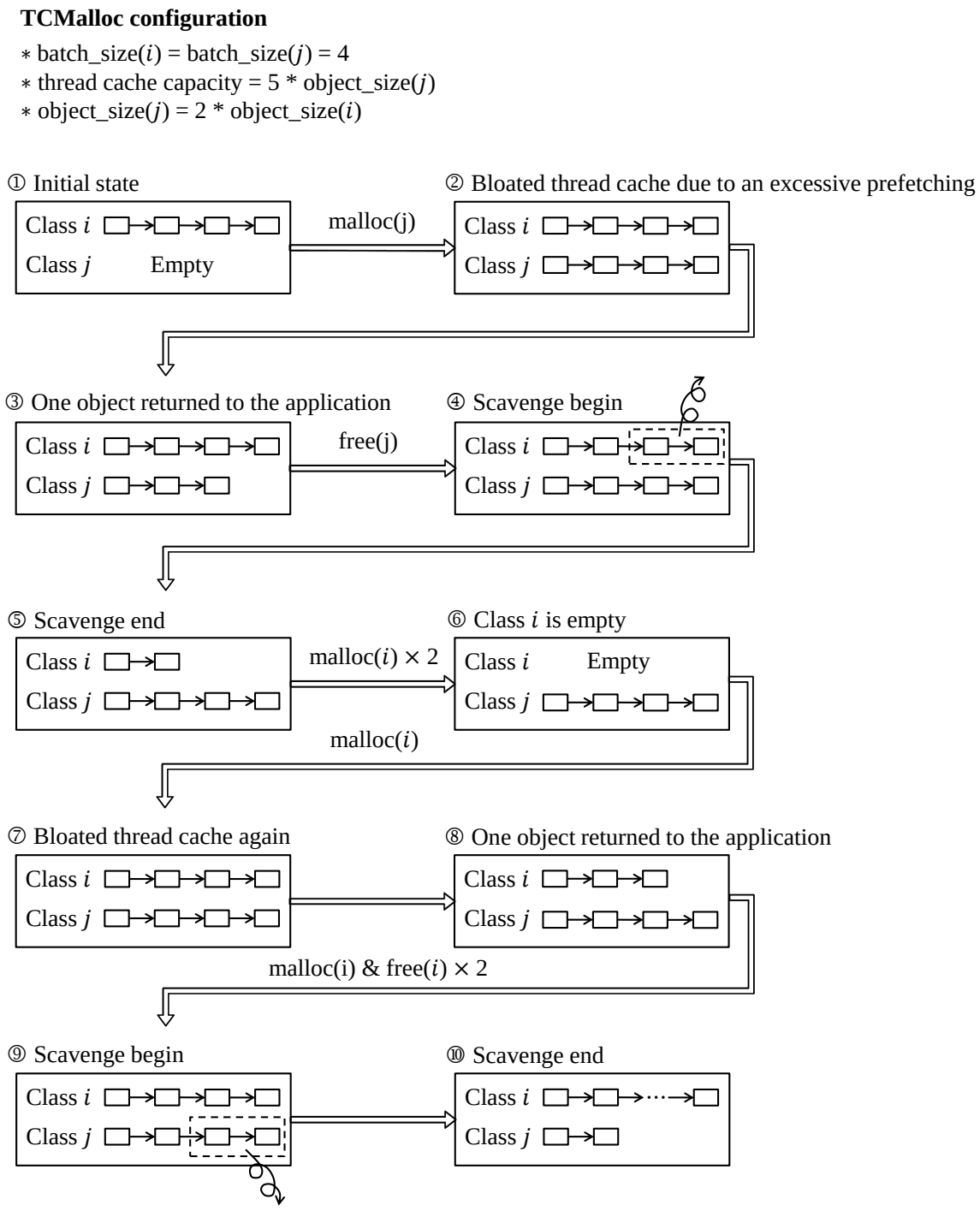

Figure 17: Redundant scavenges incurred by excessive prefetching.

In total, TCMalloc performs four management operations in the scenario. However, not all are necessary. This is because only one object is lacking from the initial state of the thread cache for serving the memory allocations. Hence, only one prefetch

Table 5: Allocation behaviors for the synthetic benchmarks.

	Allocation test	GC test
No. of threads	16	128
Objects per allocation	1	32
Allocation size ranges (bytes)	4-64	4-2K
Maximum memory usage	1GB	64KB

of class j is sufficient if the prefetch does not bloat the thread cache and cause a scavenge. In this regard, the last three management operations juggle objects between a thread cache and the central free list simply to manage the thread cache size. As mentioned in Section 3.2, applications with large thread counts have smaller thread caches and will be more vulnerable to this effect.

This pathological behavior is a performance bottleneck for two reasons. First, the operations involving the central free list need synchronization and are more complex than the ones involving only the thread caches. This may hinder the scalability of an application if a large number of threads participate in memory allocation/deallocation. Second, when objects are flushed via scavenge, they tend to bypass the transfer cache, as the number of objects transferred is unlikely to be the same as the batch size. When TCMalloc releases the objects to their corresponding *span*s, the *span* metadata has to be read and modified. Since the metadata is not part of the application, accessing the metadata frequently perturbs cache lines containing the application's own data. The end result of this is more data cache misses and data TLB misses.

3.3 Feedback-Directed Optimization of TCMalloc

3.3.1 Motivation

As indicated earlier, the memory allocation behavior of applications influences the efficiency of thread cache management. We explain this with two synthetic benchmarks, named *allocation test* and *GC test* in this work.

Table 5 summarizes the memory allocation behaviors of the benchmarks. These

Table 6: Performance of TCMalloc on the allocation test.

Batch sizes	1	32	128	1024
Allocation Hit Ratio	14.2%	96.9%	99.1%	99.5%
Execution time (s)	20.80	13.10	12.73	12.87

Table 7: Performance of TCMalloc on the GC test.

Batch sizes	1	32	128	1024
Allocation Hit Ratio	78.6%	98.0%	97.8%	97.2%
Number of Scavenges (M)	0.54	0.94	3.04	11.37
Execution time (s)	6.97	5.20	17.36	70.77

benchmarks spawn worker threads that perform several allocations and deallocations. Each thread continuously allocates memory objects from one of the specified size classes. When the memory usage reaches the specified maximum, the threads randomly deallocate some number of objects.

The two benchmarks serve different purposes. The allocation test shows the impact of the lock contention during the prefetching. Hence, the dominant operation in this benchmark is memory allocation and the benchmark allocates single objects at a time from several small size classes until the total memory allocated hits the specified threshold. On the other hand, the GC test shows the impact of garbage collections caused by excessive prefetching. In the GC test, the total size of the thread caches is made to increase by increasing the following three parameters: the number of threads that perform the allocation, the number of size classes of the allocated objects, and the number of allocated objects. This, coupled with the specific allocation/deallocation pattern used in this test, triggers a high number of scavenges.

Table 6 and 7 show the performance of TCMalloc on the benchmarks. For the allocation test, using large values for the batch sizes reduces lock contention due to a higher hit rate in the thread cache on allocations, achieving better performance. On the contrary, using large values yields poor performance for the GC test, since doing so increases garbage collection pressure, as shown by the excessive number of scavenges incurred by the 1024 batch size, despite a high thread cache hit rate.

These results clearly show that application behavior heavily affects the optimality of the batch sizes and that using static values across all applications is suboptimal. However, for the same reason, the results show that determining proper values for the batch sizes is challenging for the developers.

One remedy to this problem is to use feedback-directed optimization (FDO) [58]. In FDO, an application is run with a representative input to gather statistics that summarize the dominant behavior of the program. The statistics are then used in optimizing a binary. As such, FDO is suitable for optimizing the batch sizes by reflecting allocation behavior of an application. Traditional FDO profiles the weights of basic blocks in a program and the control flow edges connecting them, via compiler-inserted instrumentation [58]. Additionally, some values of program variables and method arguments may be profiled. To optimize libraries such as TCMalloc, we built prototype support for user-guided value profiling, called Feedback-Directed Library Optimization (FDLO). The prototype support allows the user to install hooks to user-written routines that monitor specified values at runtime, and post-process the values on exit via a custom *at_exit* handler. The installed *at_exit* handler then identifies macros that can be set during the feedback optimization compile to customize the parameter values. The prototype FDLO support was implemented on top of gcc 4.7.

As our focus is on optimizing TCMalloc, the entity that needs to be observed during the profiling run is the memory allocator itself, rather than a target application. With this in mind, the rest of this section discusses how to profile TCMalloc and presents a heuristic that uses the profile, with an objective of minimizing the number of access to the central free list by optimizing the batch sizes.

3.3.2 Profiling TCMalloc

The FDLO of the thread cache management has three goals. First, the batch sizes have to be large enough to reduce the number of access to the central free list during prefetching. Second, the batch sizes should not have extreme values that make prefetching excessive. Third, the batch sizes should be bounded by reasonable values so as not to counteract the purpose of the transfer cache.

To achieve these goals, the profiling gathers several statistics: the average of the thread cache sizes after scavenges, as well as the thread cache hit and miss counts for each size class, where thread cache hit count is defined as the number of memory allocations that were satisfied from the thread caches directly without a prefetch. These statistics are tracked by counters added to the TCMalloc routines. The first statistic is to estimate the thread cache budget that TCMalloc is willing to reserve for a thread cache. The latter two statistics are to predict the effect of increasing the batch sizes. Reducing thread cache miss is a major goal of the optimization.

There is one caveat in gathering the statistics. Specifically, the batch sizes affect the memory allocation behavior during the profiling run. For example, the default batch sizes (typically 32 objects) may obscure opportunities for improving performance from smaller batch sizes. As such, measuring the statistics as conservatively as possible is an important issue. To address this, we run the application with all batch sizes set to 1. Doing so makes the estimate for the thread cache budget most fine-grained as the thread cache is least bloated when a scavenge happens.

3.3.3 Iterative Thread Cache Space Apportioning

Figure 18 is the pseudocode of the heuristic for the FDLO *at_exit* handler. The heuristic is essentially an iterative procedure that distributes thread cache budget to appropriate size classes until the budget depletes. Initially, the batch size for each size class is set to 1. At each iteration, the heuristic computes the expected merits

```
1: procedure ComputeBatchSizes
2:     budget ← The average of thread cache sizes after scavenges
                during the profile run
3:     for each size class s do
4:         batch_size[s] ← 1
5:         TC_hits[s] ← thread cache hit count during the profile run
                        for size class s
6:         TC_misses[s] ← thread cache miss count during the profile run
                          for size class s
7:     end for

8:     while budget > 0 do
9:         for each size class s that is not finalized do
10:            if batch_size[s] ≤ 4 then
11:                next_batch_size[s] ← batch_size[s] * 2
12:            else
13:                next_batch_size[s] ← batch_size[s] + 4
14:            end if
15:            r ← next_batch_size[s] / batch_size[s]
16:            d ← next_batch_size[s] − batch_size[s]
17:            space_requirement[s] ← size(s) * d
18:            expected_TC_misses[s] ← TC_misses[s] / r
19:            additional_TC_hits[s] ← TC_misses[s] − expected_TC_misses[s]
20:        end for

21:        // Rank unfinalized size classes
22:        Sort(unfinalized size classes,
             compare=additional_TC_hits / space_requirement)

23:        s ← selected the size class with the highest rank
24:        if expected_TC_hit_ratio(s) < HitThreshold then
25:            batch_size[s] ← next_batch_size[s]
26:            TC_misses[s] ← expected_TC_misses[s]
27:            budget ← budget − space_requirement[s]
28:        else
29:            Finalize size class s
30:        end if
31:    end while
32: end procedure
```

Figure 18: A heuristic for determining proper batch sizes.

of increasing the batch sizes for each size class. The size classes are then ranked according to their respective merits. The heuristic assigns thread cache space to the

size class with the highest merit by increasing the batch size for the selected size class.

The heuristic defines the merit as the additional number of thread cache hits, which is expected from increasing the batch size, per one byte. Hence, the metric represents the relative effectiveness of assigning a thread cache space to a particular size class. Calculation of the merit is based on an assumption that doubling the batch size halves the miss rate; e.g., for 10 continuous allocations, fetching two objects in a batch results in 5 thread cache misses, while fetching only one results in 10 misses.

Considering the thread cache miss reduction is only half the picture. Increasing the batch size as long as it does not incur unnecessary scavenge reduces the prefetching overhead. However, because doing so ignores the presence of the transfer cache, it can amplify the garbage collection overhead. Between the two garbage collection methods, list truncation is influenced by the batch size via the mechanism that adjusts the max length. When the batch size is too large, the max length also becomes too large after the slow-start. Hence, there is a danger that garbage collections mostly work through scavenge. Since list truncation leverages the transfer cache while scavenge mostly bypasses the transfer cache, making excess objects flushed through list truncation is preferable. For this reason, the heuristic limits the batch size to become too large if the expected thread cache hit ratio reaches *HitThreshold* [2].

3.4 Evaluation

In this section, we demonstrate that the proposed heuristic achieves the targeted goals through the synthetic benchmarks. Additionally, we use Google internal benchmarks to show the potential benefit of applying the technique to real-world applications.

[2]We set this to 99%.

3.4.1 Implementation

As described in Section 3.3, we built a prototype user-guided value profiling infrastructure for Feedback-Directed Library Optimization (FDLO) on top of GCC 4.7. The prototype utilizes several added built-in extensions to allow the user to specify which values to profile, and to install an *at_exit* handler to post-process the collected value profiles at the end of the profiling run. Another built-in extension allows the *at_exit* handler to record macro values into GCC's profile feedback file, which are then extracted and automatically added as -D flags during parsing in the profile optimized build.

As mentioned in Section 3.3.2, during profile collection the batch sizes are artificially set to 1 to enable the finest-grain measurements of thread cache budget. The FDLO built-in extensions were used to collect the statistics described in that section. Finally, the method described in Section 3.3.3 was then used to estimate the best batch size for each size class based on the collected statistics, which were then recorded in the profile feedback file as macro values. TCMalloc was also modified to allow these macro values to override the default batch sizes for each size class.

The FDLO profiles were collected along with traditional FDO profiles via compiler-inserted instrumentation and both the profiles were used together to generate the optimized binary. Results were compared to binaries generated without FDLO, but with regular FDO and all other optimizations.

3.4.2 Synthetic Benchmark Results

The experimental results for the microbenchmarks were obtained by running the benchmarks on a machine with a six-core Intel Xeon processor having 32GB of RAM. Each configuration was run 3 times and the numbers presented are the average across all 3 runs.

Table 8 shows the performance of TCMalloc with FDLO on the allocation test.

Table 8: Performance of TCMalloc on the allocation test.

Batch sizes	1	32	128	1024	FDLO
Allocation Hit Ratio	14.2%	96.9%	99.1%	99.5%	98.8%
Execution time (s)	20.80	13.10	12.73	12.87	12.74

Table 9: Performance of TCMalloc on the GC test.

Batch sizes	1	32	128	1024	FDLO
Allocation Hit Ratio	78.6%	98.0%	97.8%	97.2%	97.4%
Scavenge count (M)	0.54	0.94	3.04	11.37	0.70
Execution time (s)	6.97	5.20	17.36	70.77	4.57

FDLO chooses large enough values for the batch sizes (96 for each of the first five size classes) and matches the best performance of the static batch sizes.

Table 9 shows the performance of TCMalloc with FDLO on the GC test. Compared to the over-aggressive configurations, FDLO reduces the garbage collection pressure by not crossing the thread cache budget. Hence, the number of scavenges does not explode with FDLO. Also, FDLO makes the prefetching aggressive enough with the selected batch sizes typically ranging between 16 and 64 for the various size classes. By balancing the thread cache hit ratio against the scavenge frequency, TCMalloc with FDLO achieves the best performance.

3.4.3 Google Internal Benchmark Results

The proposed technique was also evaluated on two important real-life applications at Google that have significant numbers of threads. The experimental results on these applications show that the problem of excessive scavenges is seen on real-life applications and mitigated by FDLO.

3.4.3.1 BigTable

BigTable is a distributed storage system for storing massive amounts of structured data [20]. We evaluated the performance using a performance benchmark for BigTable, run on a dual-processor server with two eight-core Intel Xeon processors having 64 GB of RAM. BigTable was configured with more than 400 threads and incurs a significant

overhead from frequent scavenges.

Using FDLO, batch sizes ranging from 1 to less than 100 were selected, with only the smallest size classes using batch sizes over 10. This resulted in over 10% improvement in execution time, owing to 85% fewer scavenges and 2.5% more thread cache hits.

3.4.3.2 Content Ads Targeting

The content ads targeting application selects relevant ads to show on websites containing related content and is performance-critical. Performance was evaluated using a single server running on a dual-processor machine with two eight-core Intel Xeon processors having 64 GB of RAM, processing queries sent by a load test job running on a separate machine. The application contains over 60 threads and also incurs scavenges. FDLO selects batch sizes ranging from 10 to over 1000, with several smaller size classes assigned batch sizes above 100. These batch sizes are larger than those selected for BigTable due to the relatively smaller count of threads. With FDLO, the application achieved around 1.18% improvement in throughput, owing to a 16% reduction in scavenges. The reduction in scavenges had the side effect of reducing the dTLB misses incurred by accesses to the central cache and therefore span metadata, which was confirmed by hardware counter measurements.

3.5 Summary

Optimizing memory allocator performance is still an important issue, as memory allocation is a performance hotspot for many applications. Especially for large-scale multi-threaded applications, memory allocation can be the most critical reason for scalability impediment.

In light of this, this work presents a heuristic for use in feedback-directed optimization of the popularly used TCMalloc. The proposed heuristic makes the thread

cache management in the memory allocator as aggressive as possible while preventing a performance bottleneck caused by excessive prefetching. The end result of the optimization is a reduction in the number of access to the central heap. As TC-Malloc serves the same allocation needs with less locking overhead and with less data cache and data TLB perturbation, the overall performance of an application improves with the proposed optimization. Empirical results show that up to 10% performance improvement can be achieved on Google internal benchmark applications using this approach.

CHAPTER IV

RECYCLING DEAD OBJECTS FOR REDUCING MEMORY BLOAT IN JAVA APPLICATIONS

4.1 Introduction

Memory bloat in Java applications is a well known problem. The culture of object-oriented programming in Java, combined with implicit memory management through garbage collection, encourages developers to program in a way that applications create many more memory objects than is necessary [47, 69]. As a consequence, many Java applications are developed with almost no consideration about the memory efficiency of the applications. Unfortunately, when such applications are deployed for production, they may use an excessive amount of memory than the applications ever need and may experience performance issues. This phenomenon is referred to as memory bloat.

Memory bloat affects application performance in three ways. First, object creation in Java is not free. Whenever an application creates an object, the application has to call a memory allocator and perform an initialization by invoking a constructor method of the allocated object. Even though the memory allocators for Java runtime tend to be highly efficient in handling memory allocations, the cost of memory allocation and object initialization combined can have non-negligible overhead. Second, memory bloat causes memory layout problems. Having a large memory footprint can degrade memory latency by increasing the working set size and by affecting hardware cache performance. Third, memory bloat may also lead to higher garbage collection pressure. Spending more time for garbage collection can impact application performance by introducing longer and more frequent application pauses.

The latest advances in static shape analysis [22, 23, 36] and program optimizations address the memory bloat issue by identifying the last use points of allocated objects. By definition, these points can be used as the safe-deallocation sites of the allocated objects. In fact, previous approaches to solve the memory bloat problem in Java applications add an explicit memory management mechanism to `free` the objects at their respective safe-deallocation sites. Unfortunately, this approach necessitates a specific type of memory allocator for the deallocation mechanism to work.

In this work, we take a different approach by adding an object caching layer on top of the Java memory allocation system instead of modifying existing memory allocators, which can incur another performance problem. Our approach, like previous research, uses a static shape analysis to identify the last use points or the safe-deallocation sites of allocated objects. However, our solution stores dead objects at these program points inside per-allocation site reuse caches and recycles them at the next allocation, instead of sending a memory allocation request to the memory allocator.

The object recycle optimization we propose for the memory bloat problem in Java applications is a feedback-directed optimization. The optimization performs a static shape called reference uniqueness analysis [23] to discover the safe-deallocation sites of allocated objects and to instrument the bytecodes of an application. After this step, the recycle optimization runs the instrumented application on a sample input that the developers provide for testing and optimization. Using the analysis results and the profile data, the recycle optimization selects a set of allocation sites that may benefit from the caching approach. The selected allocation sites then go through code transformations so that the application consults an object cache before creating a new object from the sites. After synthesizing reset methods for object recycling and performing an inlining optimization for the removal of cross-object invariants, the object recycle optimization finishes.

Empirical results on the DaCapo 2006 benchmark suite show that the proposed object recycle optimization has the potential for improving application performance. On the benchmark applications, the recycle optimization improves the execution time by up to 10%. Moreover, the results show that the overheads of the static analysis and the dynamic profiling are not prohibitive for real-world usage.

The rest of this chapter is structured as follows: Section 4.2 presents the motivation of this work. Section 4.3 describes the object recycle optimization technique, and Section 4.4 evaluates the performance of the proposed optimization technique on the DaCapo 2006 benchmark applications. Finally, Section 4.5 summarizes the lessons learned from the work.

4.2 Motivation

Figure 19 shows an example that we took from *bloat* in the DaCapo 2006 benchmark suite [11]. This piece of code traverses Java bytes codes to perform a number of optimizations and analyses using the visitor pattern. In this example, line 7 and line 17 create visitor objects using anonymous classes. Clearly, objects created from these allocation sites have disjoint lifetimes as the visitor objects are used only as auxiliary data structures.

This example is a typical case of memory bloat. The programmer blindly applied a widely known design pattern to implement a common task without caring about the implication on application performance. Unfortunately for this benchmark application, the creation of the visitor objects lies in the critical path. Consequently, too many objects are allocated even when their lifetimes are entirely disjoint. For this reason, the application suffers from memory bloat as it translates to higher overheads due to an excessive number of memory allocations/deallocations as well as a higher garbage collection pressure.

The solution to this problem is obvious. Instead of creating new objects every

```
1 public class FlowGraph extends Graph {
2     ...
3     private void insertProtStores(Block block,
4             HashSet tryPreds, final ResizeableArrayList defs) {
5         final Tree tree = block.tree();
6         // Visit all LocalExprs in block. ...
7         tree.visitChildren(new TreeVisitor() {
8             public void visitLocalExpr(LocalExpr expr) {...}
9         });
10
11        if (tryPreds.contains(block)) {
12            for (int i = 0; i < defs.size(); i++) {
13              LocalExpr expr = (LocalExpr) defs.get(i);
14
15              if (expr != null) {
16                  final Stmt last = tree.lastStmt();
17                    last.visitChildren(new TreeVisitor() {
18                      public void visitExpr(Expr expr) {...}
19                    });
20                }
21                ...
22            }
23        }
24    }
25 }
```

Figure 19: The relevant parts of the codes for *bloat* from the DaCapo 2006 benchmark suite that can cause a memory bloat problem.

time, we can recycle previously created objects to reduce the impact of memory bloat. Obviously, this is easier said than done. We attempt to provide an approximate solution to this problem.

Figure 20 is another example that showcases the problem of memory bloat. This example code performs a duplicate removal by using a hash set data structure. First, the driver method creates a list data structure and populates it with random numbers. Then, the code calls *removeDuplicates* to eliminate duplicates in the generated list data structure. The *removeDuplicates* method scans the input list and adds an element to the hash set if that element has not been observed yet. Finally, unique

```
 1 class Example {
 2     List removeDuplicates(List input) {
 3         HashSet seen = new HashSet();
 4         List result = new ArrayList();
 5
 6         for (Integer i : input) {
 7             if (seen.contains(i)) {
 8                 continue;
 9             }
10             seen.add(i);
11             result.add(i);
12         }
13
14         verifyResult(result);
15         return result;
16     }
17
18     void driver() {
19         for (int seed = 0; seed < 100; ++seed) {
20             List randomNumbers = new ArrayList();
21
22             generateRandomNumbers(seed, randomNumbers);
23             List uniqueNumbers
24                     = removeDuplicates(randomNumbers);
25             printContents(uniqueNumbers);
26         }
27     }
28
29   ...
30 }
```

Figure 20: A sample code that has a memory bloat problem.

elements are returned in a new list data structure and the driver method calls *printContents* to do some operations on the result.

Unfortunately, this naive implementation of *removeDuplicates* is far from being optimal. Each time we execute *removeDuplicates*, the code creates new data structures. However, the lifetimes of the data structures created from the allocation sites are completely disjoint. The hash set does not escape out of *removeDuplicates* and becomes unreachable after *removeDuplicates* returns. Similarly, the list data structure holding the resulting unique elements becomes unreachable after a loop iteration

inside the driver method. Due to this disjointness of object lifetimes, creating single instances for the hash set and the list is sufficient. Whenever we need a new instance of these data structures, we can reset the contents of already existing instances and forgo the overhead related to allocating a new object.

We should also note that *removeDuplicates* is a performance hotspot, as it is called from the driver inside a loop. Any overhead in a performance hotspot can accumulate and cause a performance issue.

Figure 21 shows how the example code can be optimized through object recycling. Since we have 3 allocation sites from where objects with disjoint lifetimes are created, *ReuseCache* class contains 3 static references for each allocation site in the example code. Each static reference points to a data structure instance that can be used during the execution. The objects contained in *ReuseCache* are collection objects that can be reset through their *clear* methods. So, each time we visit a corresponding allocation site, we return an object in *ReuseCache*, reset the returned object, and forgo an object allocation. On an experiment system (Intel(R) Xeon(R) X5550 processor with 6GB RAM running Java Hotspot(TM) Server VM on Linux), we observed 7% reduction in execution time with the optimized code.

Object recycle optimization can yield meaningful results for three reasons. First, instantiated objects go through initialization through constructor calls. If we reuse the allocated object instances, we may be able to avoid some of the costs related to the initialization. Figure 22 is an example showing this. An *ArrayList* object creates an array object in the constructor to store array elements. If we reuse *ArrayList*, we can also reuse the array object and forgo the allocation/initialization of the array object. Second, reusing allocated objects can reduce the stress on the memory system by using a lesser amount of memory space [10]. Third, by using less memory, this may lead to lower garbage collection pressure and to less application pause due to garbage collection [10].

```
 1 class ExampleOptimized {
 2     List removeDuplicates(List input) {
 3         HashSet seen = ReuseCache.slot1;
 4         seen.clear();
 5         List result = ReuseCache.slot2;
 6         result.clear();
 7
 8         for (Integer i : input) {
 9             if (seen.contains(i)) {
10                 continue;
11             }
12             seen.add(i);
13             result.add(i);
14         }
15
16         verifyResult(result);
17         return result;
18     }
19
20     void driver() {
21         for (int seed = 0; seed < 100; ++seed) {
22             List randomNumbers = ReuseCache.slot3;
23             randomNumbers.clear();
24
25             generateRandomNumbers(seed, randomNumbers);
26             List uniqueNumbers
27                     = removeDuplicates(randomNumbers);
28             printContents(uniqueNumbers);
29         }
30     }
31
32   ...
33 }
34
35 class ReuseCache {
36     static HashSet slot1 = new HashSet();
37     static ArrayList slot2 = new ArrayList();
38     static ArrayList slot3 = new ArrayList();
39 }
```

Figure 21: An optimized code for the example code in Figure 20.

For these reasons, object recycle optimization is a promising technique for automatically improving application performance. Unfortunately, object recycle optimization is a very difficult problem due to the following challenges.

```
1 class ArrayList {
2     public ArrayList(int initialCapacity) {
3         super();
4         ...
5         this.elementData = new Object[initialCapacity];
6     }
7
8     public ArrayList() {
9         this(10);
10    }
11 }
```

Figure 22: Constructors for ArrayList class.

1. Figuring out whether an object instance can be recycled or not is hard.

2. Determining which objects may actually lead to performance improvement when reused may be infeasible.

3. Transforming the codes without incurring an overhead is difficult.

Our work attempts to address these challenges.

4.3 Object Recycle Framework

In this section, we describe our object recycle optimization. The framework performs the optimization in three stages. First, it performs a shape analysis to find the beginning and end of object lifetimes. Then, the optimization performs a profiling to determine suitable optimization targets among the object allocation sites. Finally, the optimization transforms the original bytecodes to perform object reuse by recycling object instances from the selected allocation sites.

4.3.1 Overview

Figure 23 shows a high-level view of how our object recycle optimization works. This optimization method is basically a form of feedback-directed optimization in that it gathers relevant data from a profiling run of an application and uses it for the optimization. Initially, the object recycle optimization is given a set of Java bytecode

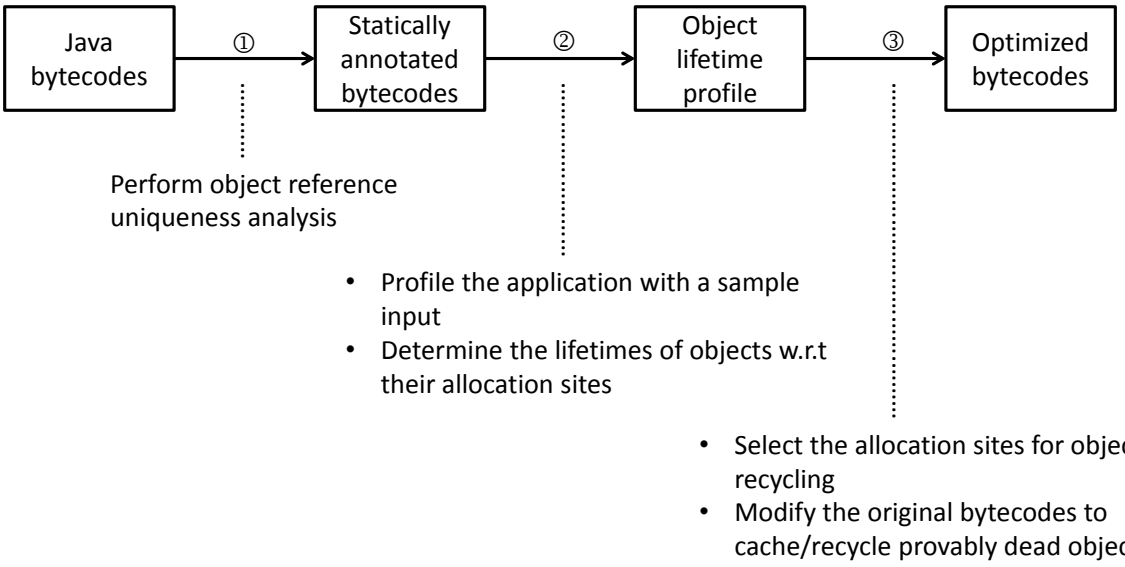

Figure 23: High-level view of the object recycle optimization.

files for which we want to apply the optimization. The optimization method performs a heap escape analysis and an object reference uniqueness analysis initially developed by Cherem and Rugina [22, 23] to discover the safe-deallocation sites of allocated heap objects.

Unfortunately, existing static shape analyses are limited in that they cannot find all deallocation sites. This is partly due to the way Java applications are written. Because the applications we target are not written in a way that is amenable for static deallocation, a statement inside a Java method may conditionally deallocate an object depending on whether the object has a single incoming reference to it or not. This may cause a problem for the caching mechanism as a cache may never be filled with dead objects during the execution. To address this issue, the object recycle optimization leverages a dynamic profiling to gather relevant information such as how much of the allocated objects from a single allocation site could have been deallocated through the identified safe-deallocation sites.

To perform this profiling, the object recycle optimization uses the static analysis results and inserts instrumentation codes for profiling. The instrumentation codes

mark the beginnings and the ends of the lifetimes of allocated heap objects based on the last use points of unique object references. The recycling framework then runs the instrumented application for a profiling and gathers object lifetime information (i.e., disjointness of lifetimes) for suitability/profitability of object recycling. With the profile data, the object recycle optimization selects a set of allocation sites for code transformation.

The code transformation for the object recycle optimization works as follows. First, the recycle transformation creates allocation site-specific object caches for the selected allocation sites. Second, the selected allocation sites are transformed so that the constructed caches are consulted first before allocating a new object during the execution. Third, the transformation synthesizes a reset method if the class definition of an allocated object does not provide a predefined reset method. Lastly, the recycle transformation performs an inlining for optimization of cross-object invariants, e.g., an instance field that holds a constant integer value throughout the lifetimes of all objects created from a single allocation site.

4.3.2 Heap Escape Analysis

The first step in our object reuse optimization is a heap escape analysis. We use an extended version of the heap escape analysis initially developed by Cherem and Rugina [22]. This analysis is a unification-based, context-sensitive escape and effect analysis that computes lightweight method summaries iteratively. The result of this analysis is the enabler of fast/scalable object reference uniqueness analysis that finds the safe-deallocation sites of allocated objects. Instead of describing the formal definition, we illustrate how the analysis works by showing an example. For a full description of the analysis, we encourage interested readers to consult the original paper by Cherem and Rugina [22].

Figure 24 shows how the method effect signature for the method *removeDuplicates*

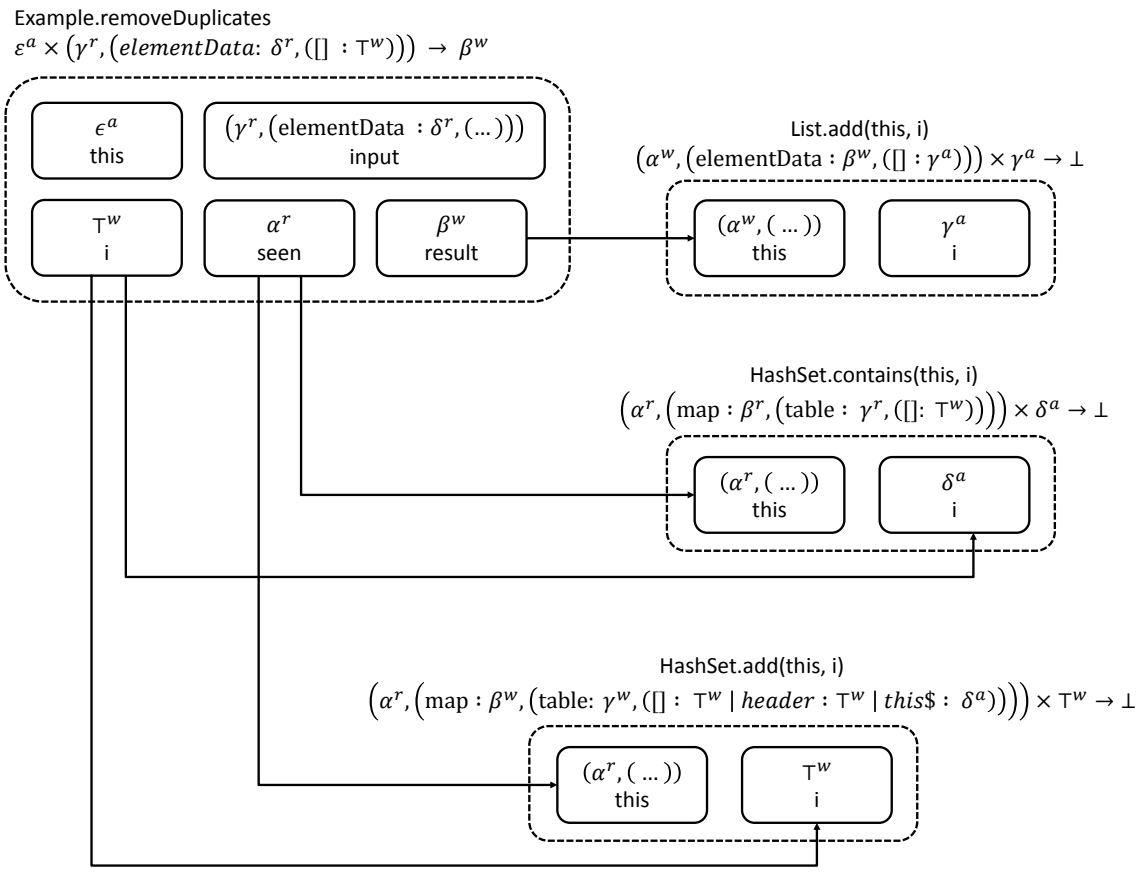

Figure 24: A method effect signature computation example.

is generated by the heap escape analysis. The method *removeDuplicates* has two parameters including the receiver object parameter, i.e., *this*, and a return value. For this method, the effect signature is initially written as follows:

$$removeDuplicates : (\epsilon^a) \times (\gamma^a) \rightarrow (\theta^a)$$

Each Greek letter in the signature is an attribute that correspond to a reachable object. For example ϵ represents the receiver object and θ represents the return object. An attribute has an access type specifier. An attribute with a represents that the reference of the corresponding object may be read. Similarly, r and w access type represent that an object field may be read or written inside the method of the signature. Additionally, the attributes can have sub-attributes that represent field

accesses as follows:

$$removeDuplicates : (\epsilon^a) \times (\gamma^r, (elementData : \zeta^a)) \rightarrow (\theta^a)$$

In this case, the second parameter object has a field named *elementData*. The effect signature represents that the field can be read inside *removeDuplicates*.

With this initial signature, the heap escape analysis first assigns fresh attributes to each reference typed local variable. Then the summary computation visits each statement in *removeDuplicates* and performs attribute unification [1].

The method *removeDuplicates* does not have any assignment statement that incurs unification between two attributes. Hence, no unification for modeling an assignment occurs and no attribute is shared as a result. However, *removeDuplicates* calls other methods and the effects of calling these methods have to be incorporated. The analysis handles this by unifying the attributes of the actual parameters with the cloned attributes of the formal parameters of the callee method. For example, the *add* method of ArrayList class takes an object and stores it inside an internal array. The method signature for *List.add* in Figure 24 captures this behavior exactly. The receiver object is represented by α and it has a field named *elementData*. Since the field is an array, storing an object into that array is represented in the signature with a sub-attribute with field name *[]*. The attribute γ appears twice in the signature and shows that the corresponding parameter can be stored into the heap through fields *elementData* and then *[]*. After unifying with the signature of *List.add*, the attribute for i remains the same. However, the access type of the attribute for *result* changes to w as an object may be stored to the heap through an access path reachable from *result* [2] [3]. The return attribute of *List.add* is the bottom element, which represents a non-reference type return value. It has no effect on the method signature.

[1] Due to the unification process, this analysis is flow-insensitive

[2] The sub-attributes of *result* and *seen* are omitted for simplicity

[3] The internal array may need to be resized. This is done by assigning a new array to *elementData*.

For another example, *Hash.add* has a more complex method signature. The first thing to note is the top element for the attribute for the second parameter. The method signatures shown in Figure 24 are 4-level effect signatures limiting the depth of the signatures to 4. Since the access path for the heap location where the second parameter gets stored is longer than 4, the attribute becomes the top element to specify that the parameter is heap-escaped and cannot be tracked any further within the limit. Upon the unification of the method signature for *Hash.add* with *seen* and i, now the attribute for i becomes the top element.

The method signature computation continues in this way and finishes when no signature changes. Since computing a method signature requires the signatures of callee methods, the effects of method calls have to propagate through call chains. The method signature computation leverages a fix-point computation for this reason.

With the computed method signatures, the effect of calling a method is summarized using the following predicates.

- $returned(m, p_i)$: This predicate is true when the attribute of the return value appears in the method signature only twice, once for the return value and one more time for a parameter. For example, the method *append(int)* in class StringBuffer returns the receiver object and the behavior is captured in the following signature.

$$\text{StringBuffer.append(int)} : \alpha \times \bot \rightarrow \alpha$$

- $stores(m, p_i)$: This predicate is true when the ith parameter gets stored into the heap. In this case, the attribute for the corresponding parameter is either the top element or appears in the signature nested under other attribute(s). For example, a *set* method can have the following signature.

$$\text{*.set(Object)} : (\alpha, (\beta)) \times \beta \rightarrow \bot$$

- $fresh(m)$: This predicate is true when the attribute for the return value appears only once in the method signature. For example, the method *toString()* in Java typically returns a new string object and has the following signature.

$$*.\text{toString}() : \alpha \rightarrow \beta$$

- $retShared(m)$: This predicate is true when the attribute of the return value appears more than once in the signature and $returned(m, p_i)$ is true. Also, if the return attribute is the top element, $retShared(m)$ is true.

Special treatment for *final* fields: One limitation of the method signature approach is the loss of precision. Because we cannot have an infinitely accurate method signature, which will be huge in size even if possible, we merge attributes if the nesting level or the number of fields of an attribute reaches a predefined threshold. Whenever this happens, the attributes become the top element, the nesting level flattens, and the branching factor of an attribute (due to instance fields) becomes 1. If we have a long call chain that makes method signatures propagate upward, any loss in precision may add up and may become a limiting factor for a later analysis. We handle one case, namely assignments into *final* fields, differently from others to partially work around this issue.

Figure 25 shows an example of this case. In this figure, we create two object instances. In the constructor for *Example2*, we pass the first object as a parameter and store it into an instance field marked *final*. Due to this assignment, the parameter object escapes into the heap and the heap escape analysis by Cherem and Rugina generates the following signature for the constructor method.

$$\textit{Example2.}\langle \textit{init} \rangle : (\alpha^w, (\beta^a)) \times \beta^a \rightarrow \bot$$

And, the predicate $stores(\textit{Example2.}\langle \textit{init} \rangle, 1)$ evaluates to true.

```
1 class Example2 {
2     final Vector vec;
3
4     public Example2(Vector v) {
5         vec = v;
6     }
7
8     public void printContents() {
9         for (Object o : vec) {
10            print(o.toString());
11        }
12    }
13
14    ...
15 }
16
17 class Driver {
18    void run() {
19        Vector v = new Vector();
20        Example2 e = new Example2(v);
21        e.printContents();
22    }
23 }
```

Figure 25: An example showing a heap escaped parameter.

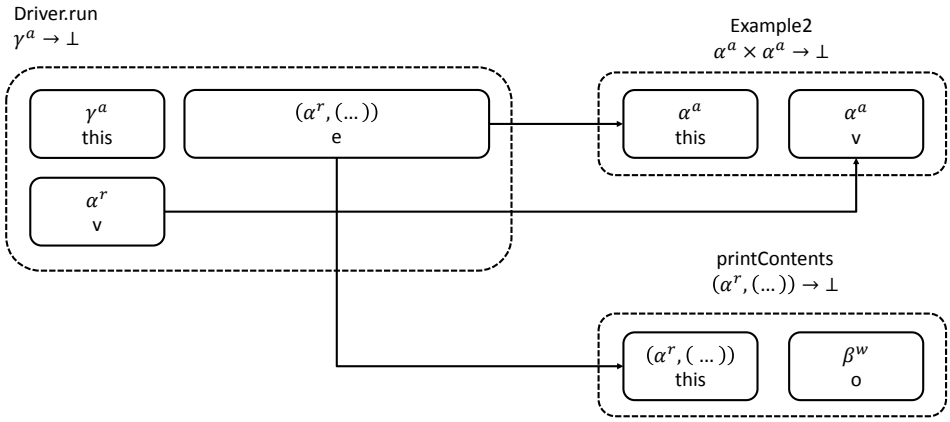

Figure 26: Assignment into *final* fields are handled differently in computing method signatures.

Figure 25 shows how we handle this case differently. First, we unify the attributes of the constructor for *Example2*. This changes the signature of *Example2* as follows:

$$\text{Example2}.\langle\text{init}\rangle : \alpha^w \times \alpha^w \to \perp$$

Then, we use this signature to update the attributes in *Driver.run*. Since the two parameters for *Example2* are unified, we also unify v and e consequently.

To handle *final* fields this way, we need another analysis to figure out which fields are *final*. Instead of relying on the presence of *final* keyword for an instance field, we perform a simple analysis before the heap escape analysis as follows. First, we initialize the set of *final* fields to be all fields we can find in the class files. Then, we analyze each method body. If the method is not a constructor method and there is an assignment into this field, we remove that field from the set of *final* fields. All fields still remaining in the set after this procedure are *final* fields.

To account for this special handling, we introduce one more predicate that a later analysis can use.

- $escapeIntoFinalField(m, p_i)$: This predicate evaluates to true when m stores p_i into one of the *final* fields.

4.3.3 Object Reference Uniqueness Analysis

The next step in our object reuse optimization is to determine the safe-deallocation points of allocated objects. For this purpose, we use a modified version of the reference uniqueness inference originally developed by Cherem and Rugina [23]. Instead of describing the formal definition, we illustrate how the analysis works by showing it through examples. For a full description of the analysis, we encourage interested readers to consult the original paper by Cherem and Rugina [23].

The object reference uniqueness analysis consists of a local flow-sensitive must-alias dataflow analysis and a global constraint-based refutation system. First, the analysis visits each method to compute flow-sensitive must-alias sets. If a variable holds a unique reference to an object, then the variable is said to be unique and the last use point of the variable can be used as a safe-deallocation point. To incorporate

1 **void** run(Vector v) {		$\{v\}^v$
2 Example2 e = **new** Example2(v);	$\{e\}^\perp$	$\{v\}^\top$
3 e.printContents();	$\{e\}^\perp$	
4 }		
	(a)	(b)

Figure 27: Flow-sensitive must-alias dataflow analysis of method run. The alias sets at each line show the aliased references for the program point after each statement. Each alias set has a tag representing a condition on the uniqueness of the references contained in an alias set. The columns (a) and (b) show the alias sets for the new object allocated inside the method and for the parameter object respectively.

the effects of calling methods during this intra-procedural must-alias set computation, the uniqueness analysis uses the method effect predicates defined by the heap escape analysis. If an object passed as a method parameter escapes into the heap, then the uniqueness analysis marks the uniqueness of the parameter variable as intractable. After this must-alias set computation, the uniqueness analysis generates global constraints that express conditional uniqueness of parameter variables and instance fields. Initially, every method parameters and instance fields are assumed to be unique. Whenever the analysis finds shared object references at method boundaries (method calls, method entry and exit points), the uniqueness of associated parameters and instance fields are refuted. Any conditional uniqueness of object references adds an implication constraint to the refutation system. After all constraints are gathered, the uniqueness analysis solves these constraints to prove or disprove the uniqueness of method parameters and instance fields.

Figure 27 shows how the local must-alias analysis works. In this example, we have two reference variables: e and v. Since *run* does not have any branch instruction, the flow-sensitive must-alias analysis finishes in a single scan from line 1 to line 3. At line 2, the method allocates a new object and a reference to the object is stored in e. After invoking *printContents* on the object, the method returns.

The dataflow facts about the allocated object are recorded using an abstract set of tagged alias sets. Figure 27 shows the computed abstract sets for the program points

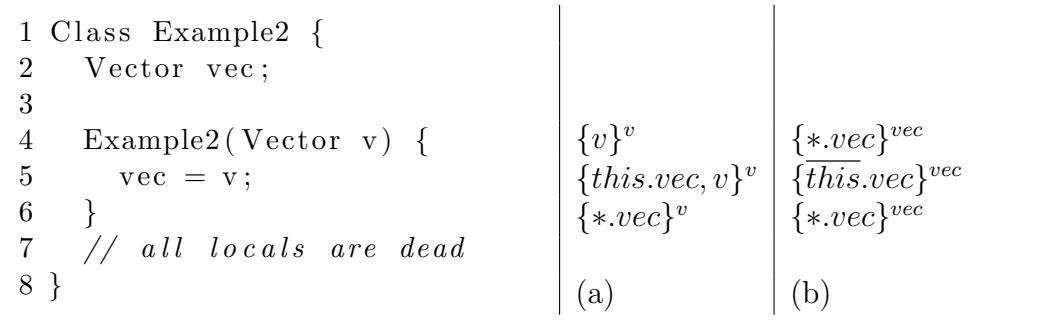

Figure 28: Must-alias dataflow analysis of a constructor method. The columns (a) and (b) show the alias sets for the parameter object and for arbitrary objects reachable through the instance field *vec*.

after each instruction. After new object allocation, we have a new alias set $\{e\}$ for the object. The tag or the superscript on an alias set represents the condition on the uniqueness of the set. For example, $\{e\}$ has the bottom element for the tag. Since e is the only reference to the allocated object, the tag on the alias set represents that the object pointed by this alias set must not alias with any other alias set (hence, an alias set with the bottom element is unique unconditionally).

The tag on an alias set can also be an instance field or a parameter object. For example, the method *run* in Figure 27 has a parameter. At the beginning of the method, the alias set for the parameter object contains the reference variable for the parameter and is tagged with the same variable. After invoking the constructor of *Example2*, the tag becomes the top element because *Example2* stores v into an instance field of e.

Figure 28 shows an example where we store an object into an instance field. Initially, at the entry to the constructor of *Example2*, the local abstraction of tagged alias sets contains two sets. The first alias set corresponds to the parameter object. The second alias set corresponds to the instance field the constructor methods access. Initially, the alias set for the instance field contains $*.vec$, which represents any object reachable through the field *vec* of any *Example2* object. After the assignment into the instance field *vec*, the must-alias analysis adds $this.vec$ into the alias set for v since

this.vec contains a reference to the same object after the assign. It is worth noting that the alias set for $*.vec$ changes to $\{\overline{this}.vec\}$ during this step. This is because the must-alias analysis partitions the set of objects reachable through *vec* into two sets, one that can be accessed through the *Example2* object involved in the assignment and the other objects reachable through *vec* the must-alias analysis does not track anymore. After the assignment statement, the constructor method returns. Since the reference variables *this* and *v* are no longer available at this point, the must-alias analysis partitions the objects for the field *vec* again. However, no reference variable of *Example2* type is live at this point. This is why the local abstraction contains only the alias sets for the field *vec*.

After all dataflow facts from the local must-alias dataflow analysis are computed, the next step in the uniqueness analysis is a global constraint-based uniqueness refutation. This process involves inspecting the results of the flow-sensitive analysis and solving the constraints.

The constraint generation happens at the method boundaries. Combining the two examples in Figure 27 and Figure 28, there are 4 method boundaries. The method exit points for the constructor of *Example2* and the *run* method constitutes 2 among the 4 method boundaries. The remaining 2 are the constructor call in *run* and the invocation of *printContents* in the same method. From the exit point of the constructor for *Example2*, we gather a constraint

$$v \notin U^* \Rightarrow vec$$

where U^* represents the set of unique method parameters and instance fields. Similarly, we gather a constraint from the must-alias set at the constructor call in the method *run*.

Figure 29 shows a graphical representation of the uniqueness constraints gathered from the examples in Figure 27 and in Figure 28. The nodes in the graph correspond to the tags of alias sets and the edges represent the uniqueness constraints among the

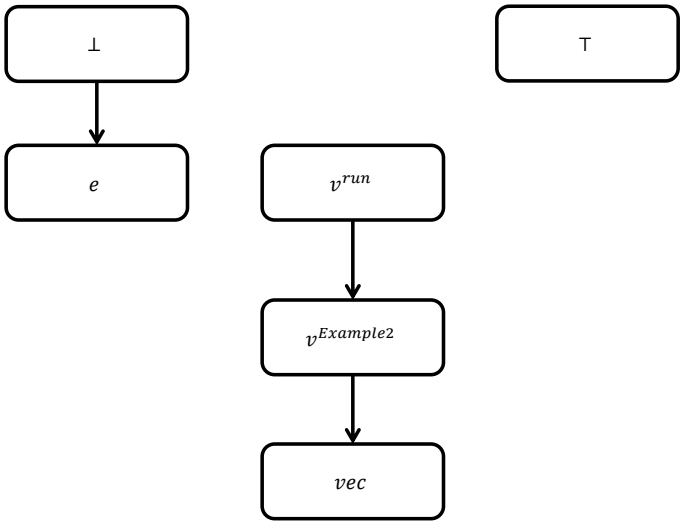

Figure 29: Uniqueness constraints represented as a directed graph. The nodes in the graph represent the tags for the alias sets from the must-alias analysis and the edges represent uniqueness constraints among the tags. Each parameter has a superscript denoting the defining method.

tags. For example, there is an edge between v and *vec* due to the constraint computed at the exit of *Example2*.

Global uniqueness constraint-based refutation is basically a reachability problem. Any tag (and the corresponding reference entity) reachable from the top element is not unique and is assumed shared. Hence, for any object reference with a shared tag, we cannot answer whether the reference is a single unique reference to the object being pointed. Solving the constraints is nothing more than computing reachability from the top element and caching the results.

After solving this reachability problem, we can now answer the question of whether a last use point of a reference variable is a safe-deallocation site or not. If the size of the alias set for the reference variable is 1 and the tag of the alias set indicates reference uniqueness, we have discovered a safe-deallocation site. In Figure 27, for example, line 4 is a safe-deallocation site for e since e is the only reference in the alias set and the tag is the bottom element. It is also worth noting that the same program point can also be a safe-deallocation point for the parameter object v. This is due to

```
 1 class Example3 {
 2     Vector vec;
 3
 4     Example3(Vector v) {
 5         vec = v;
 6     }
 7
 8     static void run(Vector v) {
 9         Example3 e = new Example3(v);
10         atAlloc(e, 1);
11         e.printContents();
12         atFree(e, 2);
13     }
14
15     static void print(Example3 e) {
16         e.printContents();
17     }
18 }
```

Figure 30: A code instrumentation example. Instrumentation calls are inserted at object allocation sites and at safe-deallocation sites.

the uniqueness of the instance field *vec*. When *e* becomes dead, the object reachable through *e.vec* loses the last reference to it due to the uniqueness of the instance field.

4.3.4 Code Instrumentation

With the reference uniqueness analysis results, the next step in our object reuse framework is to instrument the bytecodes.

Figure 30 shows an example code that the object reuse framework generates after the instrumentation. This example is an extension of the examples in Figure 27 and in Figure 28. The uniqueness of the reference variables and the parameter objects are the same as before. The parameter object *e* in *print* is also unique.

There are three things to discuss with regard to inserting instrumentation calls. First, there is a fresh object allocated inside *run*. For each reusable object allocation [4], the recycle optimization method inserts an instrumentation call *atAlloc*. The

[4]An object is reusable only if we can reset the contents of the object before reuse. Java library

instrumentation method *atAlloc* has two parameters. The first parameter is the allocated object itself and the second parameter is an instruction identifier. We assign unique numbers to each instrumentation call instruction so that we can identify the allocation site or the safe-deallocation site given an object during the profiling. The lifetime of the allocated objects ends after the call to *printContents*. The framework marks the point by inserting another instrumentation call *atFree*. Similar to *atAlloc*, *atFree* has the same parameters.

Another thing to note is that there is no instrumentation call for the unique fields of an object instance. Even though the field *vec* is unique, the instrumentation does not insert *atFree*. The granularity of object reuse in our optimization is object shape rather than each individual object. For the object referenced by the unique field *vec*, the way we reuse this object is through re-initialization. For example, if the constructor method *Example3* had an object allocation (instead of taking a parameter) and an assignment to the unique field right after, our framework uses this by resetting the contents of the object referenced by *vec*, should we opt for an object reuse for the allocation site at line 10.

The last thing to note about the instrumentation is how the instrumentation handles unique parameters. Even though the reference to the parameter object is unique in *print* method, we do not insert an instrumentation call unless the method signature of *print* indicates the parameter escapes into the heap. Instead, the instrumentation happens at the caller of this method. This is for two reasons. First, the farther away an object is from its allocation site, the greater the chance is of losing the type information. Having more precise type information helps in transforming the allocation sites and safe-deallocation sites of selected objects for object recycling. Second, as long as the application is able to reuse the object instances, the timing does not matter. Retrieving an object for recycling can happen any time if it will be done before

---

classes with no known reset method are not reusable.

a new allocation of the same object.

4.3.5 Object Lifetime Profiling

With the instrumented codes, the next step in the object reuse optimization is profiling the application.

Profiling the runtime behavior of an application serves two purposes. First, the uniqueness analysis does not compute the correspondences between allocation sites and safe-deallocation sites. To be able to recycle object instances, we have to figure out which objects are created from where and where we may safely deallocate these objects. That is, we need to know the safe-deallocation sites for an object allocation site. We approximate this information using a profiling. Second, transforming all discovered allocation sites and safe-deallocation sites may not be an optimal decision. Doing so can increase the code size unnecessarily as well as bloating the cache holding the objects for recycling. We try to avoid introducing any unwanted overheads.

Figure 31 shows how our profiling method works. We run an application with a Java programming language agent on a sample input. The agent provides an implementation of *atAlloc* and *atFree* instrumentation methods. When the application calls *atAlloc*, the agent performs two things. First, it inserts a map entry into a *WeakIdentityHashMap* so that we can query the allocation instruction of an object when we handle a *atFree* later. Second, the agent records the allocation statistics. Since the application may be multi-threaded, our profiling method keeps a list of allocation sites per each thread and merges the statistics after the application finishes the execution. Handling of *atFree* is similar. First, the agent queries the allocation site of the dead object and records the current state of the corresponding allocation site.

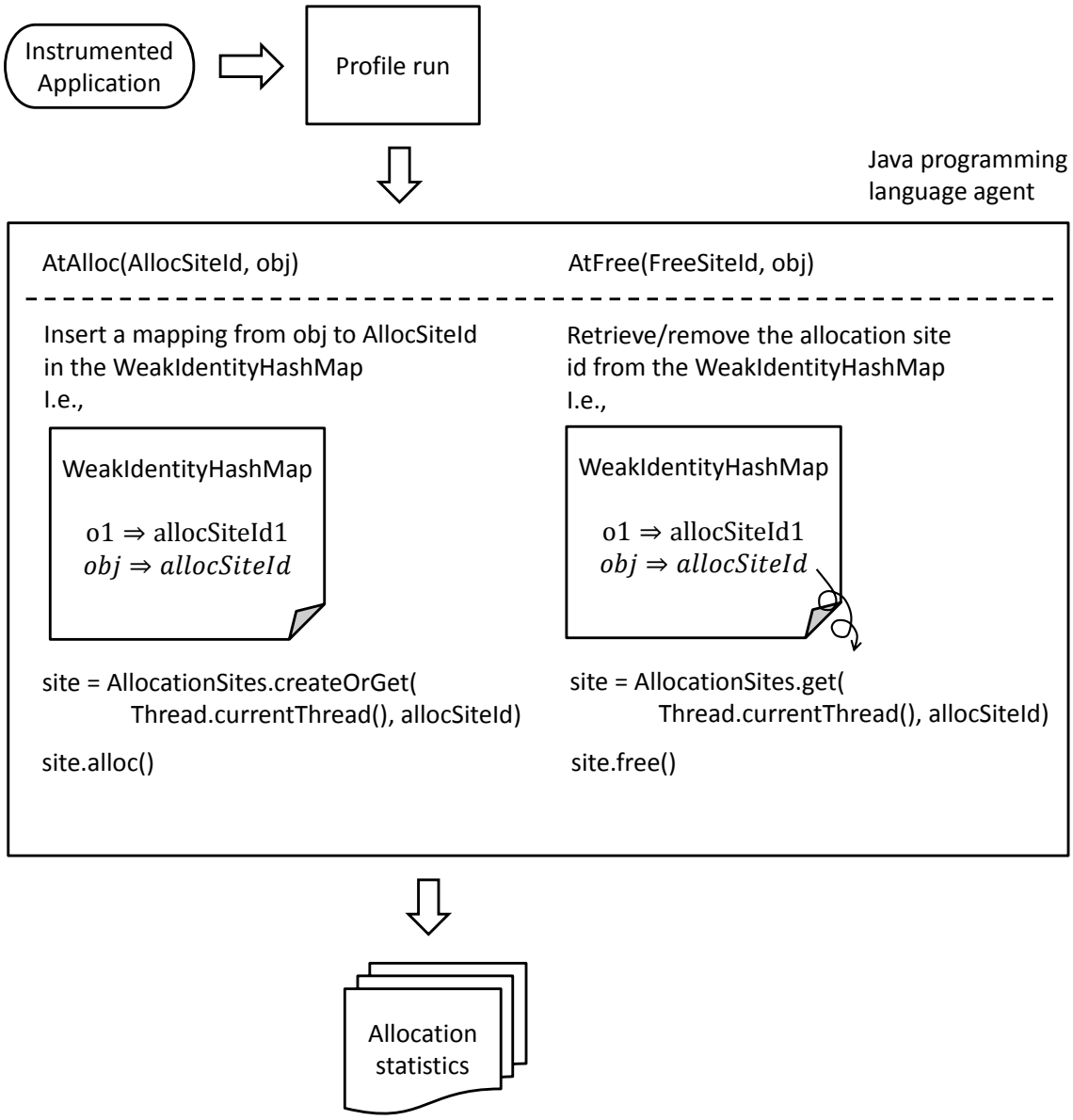

Figure 31: Profiling infrastructure for the object recycle optimization.

The *WeakIdentityHashMap* is an important data structure for associating an allocation site to a corresponding safe-deallocation site and vice versa. The *WeakIdentityHashMap* is a version of *IdentityHashMap* that uses *weak* references for the internal map entries. Since the uniqueness analysis may not be able to find all safe-deallocation sites due to its limitations, inserting every object into a map data structure lengthens the lifetimes of the inserted objects indefinitely. That is, if the static analysis

fails to discover some of the safe-deallocation points of allocated objects and the map data structure maintains the references to the dead objects, the objects will remain in memory due to their reachability. By using *weak* references we solve this problem. Whenever a reference inside *WeakIdentityHashMap* becomes the last reference, garbage collection will reclaim the object with only *weak* references according to the semantics of *weak* references.

The allocation site-specific *alloc* method gathers three statistics. First, it records the instance size through the instrumentation library provided by Java. This information is used for ranking allocation sites before selecting them for the object recycle transformation. Second, we record the safe-deallocation sites. Lastly, we check the number of objects allocated between *atFree* calls. We perform object reuse transformation only when the lifetimes of objects from an allocation site are disjoint.

With the profile data, the object recycle optimization selects allocation sites for the recycle transformation as follows.

1. An allocation site responsible for more than 1% of total memory allocations is a candidate for the optimization.

2. Alternatively, if the allocation site creates a StringBuffer or StringBuilder object, it remains in the candidate set.

3. Then, the remaining allocation sites are ranked according to:

   instance size * number of object allocated

4. After this, the top k allocation sites are selected for the transformation [5].

[5]We use a cache size threshold of 4KB. We include allocation sites for the object recycle transformation as long as we have a remaining space in the cache

```
1 class Example4 {
2     Vector vec;
3     Vector vec2;
4     int n;
5
6     Example4(Vector v) {
7         vec = v;
8         vec2 = new Vector();
9     }
10
11    static void run(Vector v) {
12        Example4 e = new Example4(v);
13        e.printContents();
14    }
15
16    static void reset(Example4 e, Vector v) {
17        e.vec = v;
18        e.n = 0;
19        e.vec2.clear();
20    }
21    ...
22 }
```

Figure 32: An example of a synthesized reset method.

4.3.6 Reset Method Synthesis

The next step in the object reuse optimization is to generate reset methods for the selected allocation sites after the profiling. Each time we recycle an object, we need to reset the contents of the object. For this purpose, the object recycle optimization synthesizes reset methods for each class that has been selected for reuse.

Figure 32 shows a synthesized reset method for an example class. Basically, a synthesized reset method is a copy of a constructor method. For example, the method *run* calls the constructor *Example4* at line 10. Assume that this allocation site is selected for object reuse. Line 14 shows the signature for the synthesized reset method for this allocation site. Unlike the original constructor, the synthesized reset method is a static method. Java VM prohibits adding an instance method that can be called

$$IN[s] = \bigcap_{p \in pred[S]} OUT[p]$$

$$OUT[s] = IN[s] \cup GEN[s]$$

$$GEN[d : this.f \leftarrow ...] = \{f\}$$

$$GEN[d : call(func, ...)] = Must\text{-}Initialized(func)$$

$$Must\text{-}Initialized(func) = OUT[exit(func)]$$

$$AllFields(class) = \{f : \text{for all fields } f \text{ defined in } class\}$$

$$May\text{-}Uninitialized(class, ctor) = AllFields(class) \ \setminus \ [Must\text{-}Initialized(ctor)]$$

Figure 33: The dataflow equations for computing may-uninitialized instance fields.

everywhere using *specialinvoke* instruction unless the instruction is for calling a constructor. To circumvent this issue, we use a static method instead.

A synthesized reset method deviates two details from the original constructor. First, every new object allocated in Java is initialized by the memory allocator before it is returned to the application. When we recycle an object, however, this is not true. So, a synthesized reset method has to make sure it does not change the language semantics. To figure out which instance fields need to be reset this way, we perform a dataflow analysis in Figure 33. Line 17 is an example of this explicit re-initialization.

Another complication in synthesizing a reset method is the handling of unique fields. For example, the field *vec2* is a unique field of *Example4*. Assume that this field is also a *final* field. Since we know *vec2* holds a unique reference and the object that gets stored into the field is allocated in the constructor, the reset method may also do the same thing. Instead of copying the allocation statement into the reset method, however, the object recycle optimization uses this information and inserts a call to a corresponding reset method.

4.3.7 Recycling Transformation

The last step in the object reuse optimization is to transform the original codes to use the cached objects. This transformation involves 1) generating a reuse cache, 2)

```
1 class Example5 {
2     Vector vec;
3     Vector vec2;
4     int d;
5
6     Example5(Vector v, int direction) {
7         vec = v;
8         vec2 = new Vector();
9         d = direction;
10    }
11    Example5(PlaceHolder p) {
12    }
13    static void run(Vector v) {
14        //Example5 e = new Example5(v, 1);
15        Example5 e = ReuseCache.getFromSlot1();
16        Example5.reset(e, v, 1);
17        e.printContents();
18    }
19    static void reset(Example5 e, Vector v,
20          int direction) {
21        e.vec = v;
22        e.vec2.clear();
23        e.d = direction;
24    }
25    ...
26 }
27
28 class ReuseCache {
29    static PlaceHolder p = new PlaceHolder();
30    static Example5 slot1 = new Example5(p);
31    static Example5 getFromSlot1() {
32        return slot1;
33    }
34 }
```

Figure 34: A code transformation example for object recycle.

replacing object allocations with get methods, and 3) an inlining optimization.

Figure 34 shows an example of a generated reuse cache and a replaced allocation expression. For each allocation site selected for object recycle, we add a static field in the *ReuseCache*. Since the allocation site in the example is a simple one that the object allocated dies before returning from *run*, we create a single object for that

allocation site in the static initializer of *ReuseCache*. Then, we replace the allocation at line 15 with a call to the corresponding get method in *ReuseCache*. After this, a call to the reset method is inserted (line 16).

This object recycle transformation creates another optimization chance. The reset method of *Example5* assigns a constant to an instance field. Each time the application invokes this reset method, therefore, it assigns the same value to the field. If the field is a *final* field, this assignment is unnecessary the next time the application resets the object before reuse.

Figure 35 is how our inliner handles this case. First, we inline the reset method into the body of the method containing the corresponding allocation site. Then, our inliner examines assignment into the fields of the object being recycled. Any assignment that has a constant value on the right hand side is removed and copied into the constructor we generate for creating a temporary object. Performing this optimization saves unnecessary re-initialization.

4.4 Evaluation

4.4.1 Implementation

The object recycle optimization is implemented using soot-2.3 (specifically, on top of the framework developed by Cherem and Rugina for their work). Since the instrumented/optimized bytecodes are emitted by soot, the bytecodes also go through the optimizations implemented in soot.

4.4.2 Analysis Overhead

One concern for using static analysis and profiling is the analysis time and the profiling overhead. If an optimization technique has an infeasible overhead, the technique is not usable. In this section, we present our experimental results on the overhead related to the object recycle optimization. All the experiments results were gathered using the system configuration in Table 10.

```
1 class Example5 {
2     Vector vec;
3     Vector vec2;
4     int d;
5
6     Example5(Vector v, int direction) {
7         vec = v;
8         vec2 = new Vector();
9         d = direction;
10    }
11    Example5(PlaceHolder p) {
12        n = 1;
13    }
14    static void run(Vector v) {
15        // Example5 e = new Example5(v, 1);
16        Example5 e = ReuseCache.getFromSlot1();
17        e.vec = v;
18        e.vec2.clear();
19        // e.n = 1;      removed after inlining
20        e.printContents();
21    }
22    ...
23 }
24
25 class ReuseCache {
26    static PlaceHolder p = new PlaceHolder();
27    static Example5 slot1 = new Example5(p);
28    static Example5 getFromSlot1() {
29        return slot1;
30    }
31 }
```

Figure 35: A reset method inlining example.

Table 10: Experiment system configuration

CPU	Intel(R) Xeon(R) CPU E3-1230 V2 @ 3.30GHz
Microarchitecture	Ivy Bridge
Level 1 cache size	4 x 32 KB 8-way set associative instruction caches
	4 x 32 KB 8-way set associative data caches
Level 2 cache size	4 x 256 KB 8-way set associative caches
Level 3 cache size	8 MB 16-way set associative shared cache
Memory	32GB
Operating System	64 bit Linux with a kernel version 3.13.0

Table 11: Analysis times for the DaCapo benchmark applications.

Application	Final Field Analysis	Escape Analysis	Uniqueness Analysis
antlr	1.12s	1.12s	5.26s
bloat	1.77s	6.81s	5.25s
chart	3.20s	6.80s	12.2s
hsqldb	2.20s	2.48s	12500s
luindex	1.64s	5.58s	3.82s
lusearch	1.63s	4.24s	3.01s

Table 12: Overheads for profiling the DaCapo benchmark applications.

Application	Slowdown factor
antlr	1.19x
bloat	12.91x
chart	4.86x
hsqldb	4.10x
luindex	1.96x
lusearch	2.34x

Table 11 shows the static analysis times for the DaCapo 2006 benchmark applications. Due to a technical reason, we were able to run our technique on 7 out of 12 applications. For most applications, the analysis time is less than 30 seconds. One notable exception is *hsqldb*. In this application, some of the must-alias sets merge even when they do not represent an alias. This happens because of the conservative handling of array accesses. Due to the enlarged alias set, the flow-sensitive must-alias analysis exploded by generating 2^n possible alias sets for n reference variables. If an application does not use array accesses heavily this behavior should not happen and the static analysis cost will not be a hindrance for the production use of the object reuse optimization. Since the cost of instrumenting the bytecodes after the static analysis is almost unnoticeable, the time for instrumentation is not included in the table.

Another concern for the overhead is the profiling time. Table 12 shows the experiment results on the DaCapo 2006 benchmark applications. For 3 applications, i.e., *antlr*, *luindex*, and *lusearch*, the slowdown factor is less than 3. Considering that the

Table 13: Performance impact of the object recycle optimization on the DaCapo benchmark applications with the default heap configuration.

	bloat	chart
Improvement in execution time	2.14%	0.97%
Reduction in objects generated	38%	4.3%
Reduction in GC invocations	7.35%	1.80%

DaCapo benchmark applications are allocation intensive and also the nature of our profiling method [6], this slowdown factor seems reasonably good. For *chart* and *hsqldb*, the slowdown factor is more than 4 and it is more than 12 for *bloat*. Although this overhead seems like too much, for the default input of the benchmark applications, the absolute time to run them does not take a prohibited amount of time. Also, none of the applications experiences a failure during the profiling. As long as a target application is not time-critical, using the profiling should not hamper the use of object reuse optimization.

4.4.3 Performance Impact

To measure the performance impact of our technique, we applied the object recycle optimization on the DaCapo 2006 benchmark suite. To establish statistical robustness, we ran each application 50 times (50 JVM invocations), where each run of the application went through 100 iterations to measure steady-state performance [7] For this experiment, we used the Java HotSpot(TM) Server VM 1.6 and ran each application with the default input provided by the benchmark suite.

Table 13 shows the performance impact of the object recycle optimization on *bloat* and *chart* with the default heap configuration [8]. Due to the object recycle, these applications create less memory objects and use less memory than they originally

[6] We maintain a map of objects to their allocation sites at runtime. This operation is invoked whenever we create a new object or when we hit a safe-deallocation site.

[7] Java applications tend to exhibit execution time variations due to various reasons. Repeating the experiments until we can achieve a statistically meaningful result is the only way to address this issue.

[8] In the default configuration, the Java virtual machine dynamically adjusts the heap size.

Table 14: Performance impact of the object recycle optimization on the DaCapo benchmark applications when the heap size is restricted to 256MB.

	bloat	chart
Improvement in execution time	10%	0.52%
Reduction in objects generated	38%	4.3%
Reduction in GC invocations	40%	3.9%

used. Also, there is some impact on the number of garbage collection invocations. Additionally, the object recycle optimization provides faster memory allocation for the allocation sites selected for recycling. With all these effects, the execution time of *bloat* reduces by 2.14% and that of *chart* by 0.97%.

Table 14 shows the performance impact of the object recycle optimization on the same applications when the heap size is restricted to 256MB. Due to the smaller memory, the JVM invokes garbage collection more frequently. The object recycle optimization is more beneficial for *bloat* in this case and has more impact on the number of garbage collections compared to the default configuration that can increase the heap size indefinitely.

4.4.4 Limitations

Unfortunately, these results are far from optimal. We discovered several shortcomings to our approach. First, there are plain-old-data-structure (POD) types that we could effectively reuse. In the context of object recycle optimization, these data structures have only public fields and do not provide a constructor for initializing the fields. Due to this, the object recycle optimization assumes that all instance fields in POD objects need re-initialization, even if that is redundant. Generally, the object recycle optimization does not properly make use of this optimization opportunity.

Second, our object recycle transformation does not involve modifying the VM. There may be more ways to improve the performance impact of object reuse if we had a better way of resetting the object contents. Currently, we had to add static methods to handle the reset needs. Any method that the inliner does not handle

remains in the final code and increases the code size. This may have an impact on application performance depending on how Java VM handles the inserted code.

4.5 Summary

Memory bloat in Java applications is an often neglected aspect of program execution. When memory bloat is benign, the impact on system performance is minimal. Unfortunately, when the bloat becomes significant, it leads to higher memory consumption and more processing time as applications spend more time performing memory allocations as well as experiencing longer and more frequent application pauses due to a higher garbage collection overhead. For this reason, mitigating the effect of memory bloat can be an important program optimization goal.

To address the memory bloat problem in Java applications, this paper presents an object recycle optimization technique. Our optimization method transforms selected allocation sites so that when an application creates an object from one of the selected sites, the application first checks the reuse cache of the allocation site and recycles a previously dead object existing in the cache. To achieve this, the optimization technique performs a static analysis to compute the safe-deallocation sites of allocated heap objects and performs a dynamic profiling to select allocation sites for object recycle transformation. After the analysis steps, the object recycle optimization technique transforms the selected allocation sites and their corresponding safe-deallocation sites to recycle provably dead objects from the selected allocation sites.

Empirical results on the DaCapo 2006 benchmark suite show that the proposed object recycle optimization has a potential for improving application performance. On the benchmark applications, the recycle optimization improves the execution time by up to 10%. Moreover, the results show that the overheads of the static analysis and the dynamic profiling are not prohibitive for real-world usage.

CHAPTER V

RELATED WORK

This chapter discusses how each source of memory bloat has been addressed independently by previous research and how this relates to our work.

5.1 Related Research: Staleness-Based Memory Leak Detection

The dominant form of memory bloat occurs through memory leaks. Notably, continued memory leaks exhaust available system memory and eventually lead to sudden hang/crash failures as a result. Since memory leak is a major threat to system reliability, this form of memory bloat has been a serious concern and a hot research topic.

In the literature, memory leak is defined in two ways. In the first view, which is usually found in the literature dealing with memory leaks in C and C++ applications, researchers approximate the liveness of allocated objects with their reachability. That is, an object is live as long as there is an incoming reference to the object. As per this view, an allocated object has *leaked* from an application only if the object becomes *unreachable* and, at the same time, if the object has not yet been returned to the memory allocator with a call to the deallocation method. In the second view, which is usually found in the literature dealing with memory leaks in Java applications, the liveness is no longer approximated with the reachability of allocated objects. In this view, even if an allocated object has an incoming reference, it can be regarded as having leaked if the object has no future use.

The first type of memory leak is easier to address as the approximated liveness information is easier to compute. Thanks to this approach, a number of static analyses

have been proposed and shown to successfully address the first type of memory leaks in C and C++ applications [37, 64, 52, 21, 42, 59]. Moreover, the reachability-based definition allows tracking memory leaks at runtime [26, 16] to locate problematic program points.

Unfortunately, the second type of memory leak is much more difficult to address as the definition here relates to the future access of an allocated object. When an allocated object becomes stored in another object and hence heap-escaped, tracking the liveness of the object becomes almost impossible to compute precisely and statically.

Staleness-based leak detection techniques were born in an attempt to address this issue. Simply put, staleness-based leak detection is a history-based future prediction in that it predicts whether or not an allocated object has a further use with previous access patterns. Object staleness that represents how long an object remains unaccessed serves as the major predictor in staleness-based leak detection.

Unfortunately, staleness-based leak detection relies too much on the user's expectation of object access patterns for its accurate operation. In other words, the leak detection technique parameterizes the staleness predicate, which specifies the degree of staleness above which an object is regarded as having leaked, as a user-configurable variable. Consequently, staleness-based leak detection needs careful user intervention for generating useful information.

The introspective memory leak detection framework this work proposes is an attempt to address this problem. Unlike previous staleness-based leak detectors, our approach is based on observing application behavior during a profile run and on constructing behavior models for the observed behaviors of heap objects. As a result, the introspective leak detection framework replaces user-configurable staleness predicates with a model-based prediction. Moreover, the leak detection framework improves the detection accuracy by leveraging additional information that has not been explored by previous research, i.e., allocation context and object coexistence patterns that the

framework observes during a profile run.

Previous research on staleness-based memory leak detection has mostly focused on implementing lightweight memory access sampling. The sampling methods can categorized in the following way. All of the sampling methods can be plugged in for use with our model-based leak detection framework and the error-prone user intervention required for the leak detection can be eliminated to achieve more precise results with fewer false positives/negatives.

Path-Biased Code Sampling: Chilimbi and Hauswirth were the first to propose the *staleness* based leak detection in their pioneering system called SWAT [24]. The *staleness* update in SWAT relies on code instrumentation of memory access instructions. To reduce the overhead, SWAT uses a path-biased sampling in tracking heap accesses. It samples each program path at a different rate. The sampling rate is in inverse proportion to the execution frequency. In this way, SWAT can reduce the overhead, since the instructions on a hot path rarely get sampled. However, the sampling can result in overestimating the staleness of the objects in hot paths, leading to false positives. Thus, the effort to reduce the runtime overhead may end up undermining the quality of the leak detection.

Page-Protection-Based Data Sampling: Novark et al. present a system called Hound that removes the heavyweight instrumentation for tracking object *staleness* using a page-level sampling [50]. The basic idea is to employ a memory protection mechanism of an OS kernel to detect the accesses of the objects. Hound periodically protects every page and updates the last access time of all objects on the same page to the *protection* time. Once a page fault occurs, Hound catches the signal and unprotects it for a performance reason; here, Hound does not update the last access time of all objects on that page until it gets protected again. That is, actual staleness updates are always delayed to the *protection* time. The resulting staleness is underestimated and this poses a risk of false negatives.

Another cause of false negatives is that Hound works at the granularity of a page; it is possible that a page contains both live and dead objects, and a single access to a live object can cause a reset to the staleness of *dead* objects in that page. To mitigate that, Hound changes the underlying memory allocator to perform an age-based segregation of allocated heap objects, which can end up degrading the performance of the memory allocator. Nevertheless, the page-level false sharing can still occur depending on memory allocation patterns.

A recent extension of Hound addresses the accuracy problem with context-aware data sampling that takes into account an allocation-call-path for the segregation of heap objects [45]. Here, heap objects are allocated in a different page according to their allocation contexts, i.e., call path to the `malloc (new)`. This approach has a better potential to get those objects with a similar access pattern allocated in the same page than a naive allocation-site-based segregation. However, no accuracy results compared to Hound's age-based segregation are presented in the paper.

ECC Protection Based Sampling: Qin et al. present a different approach called SafeMem [53]. It first groups heap objects according to their size and the calling contexts of the allocation site, and measures the lifetime of each object. SafeMem relies on the observation that the maximal lifetime of objects in the same group remains stable and is thus anticipatable. The underlying assumption is that if the lifetime of a certain object is much longer than the expected lifetime of the group it belongs to, then the object is likely to have leaked. To reduce false positives, SafeMem monitors the accesses to such suspicious objects using an ECC memory protection mechanism. That is, the heap data is scrambled and stored in the memory, and the first access to data, which is recognized by the ECC fault, leads to a conclusion that the object has not leaked.

However, such a method arrives at a premature conclusion in that an object can end up having leaked even after multiple accesses. To avoid false negatives,

SafeMem keeps watching some objects even after their first accesses by having the ECC fault handler maintain metadata such as the lifetime and its maximum of the group. Whenever an object becomes suspicious, i.e., the lifetime is longer than some threshold, the ECC monitoring is periodically enabled.

5.2 Related Research: Memory Bloat Prevention Mechanisms in Multi-threaded Memory Allocators

The performance problems due to memory bloat inside memory allocators occur due to the demands on modern memory allocators. To support systems with a large number of cores (and thus threads), modern memory allocators leverage thread-local caches of free objects for accelerating memory allocation performance. This design decision raises an issue as to how to manage the thread-local storage.

TCMalloc, the memory allocator we target in this work, manages the thread-local caches in an aggressive and eager manner. That is, when the memory allocator detects a bloated cache, it immediately flushes the cache. However, this strategy may result in the sub-optimal performance of the memory allocator, depending on the allocation behavior of an application. Our solution addresses this problem by tuning the thread cache management mechanism in TCMalloc to an observed behavior of an application. This optimization makes the object prefetching in TCMalloc aggressive until the expected benefit is offset by the cost of handling bloated thread-local caches due to the aggressiveness.

This problem has been and will likely remain an issue for memory allocators. For example, jemalloc [30] is another memory allocator that uses the same thread caching technique as TCMalloc. This memory allocator addresses the thread cache management problem in a different way by using lazy flushing. The mechanism jemalloc uses differs from that of TCMalloc in two ways. First, the flushing is periodic rather than on-demand. As such, jemalloc does not require detecting a thread cache bloat, the condition that triggers a flushing. In this respect, TCMalloc is more aggressive

in striving to reduce memory bloat due to thread caching. Second, partly influenced by the periodic flushing, the amount of object prefetching in jemalloc is dynamically adjusted. Due to these differences, the thread cache management in jemalloc is not parameterized and is not a target of an optimization.

Although not directly related, a similar issue is found in the design of garbage collectors for managed languages. For such systems, several garbage collection papers have examined the trade-off of when or how best to reclaim memory in multi-threaded systems [5, 12]. The goals of this previous research differ from those of our work in that they are limited to comparing garbage collection algorithms or to meeting real-time constraints.

The FDLO technique described in this document results in tuning the memory allocator based on application behavior. A related approach is to write application specific custom memory allocators [35] to improve a given application at the cost of reduced portability. However, Berger et al. [9] showed that for most applications, a state-of-the-art general purpose allocator performs as well or better than custom allocators and the use of FDLO is likely to further reduce the advantages of custom allocators.

5.3 Related Research: Object Recycling for Java applications

Memory bloat in Java application is a well known issue. The culture of object-oriented programming, combined with the illusion of automatic memory management through garbage collection, encourages developers to write programs in a way that creates more objects than are necessary [47]. Because this can lead to high stress on the memory system and also to high garbage collection pressure, memory bloat can have a negative impact on the performance of a Java application.

In fact, several performance tuning guides mention this problem and suggest several solutions [57, 51]. However, this is rarely practiced at the application developer

level as the source of the problem stems from a demand for increased developer productivity, which tolerates some inefficiency in the code [47]. For this reason, removal of memory bloat in Java applications has been a hot research topic.

The research performed to address the memory bloat issue can be categorized in the following way.

First, a number of approaches have focused on providing programming aids to application developers [65, 67, 66, 48, 25]. These tools are based on profiling the application during the runtime. They generate helpful information for application developers.

Second, a number of static analyses for finding temporary objects have been developed by the static analysis research community. Dufour et al. and Shankar et al. present static analyses that target the detection and removal of temporarily created objects [29, 56]. Their method is based on method inlining that makes the lifetimes of allocated objects confined to a single method. After the inlining decision, temporary objects are stack-allocatable by the Java runtime. Bhattacharya et al. also present a static analysis that detects the creation of temporary containers and string objects within a loop [10]. Their solution performs a source-to-source transformation that makes the objects reused at the loop level.

An interesting branch of research is dedicated to transforming implicit memory management into explicit memory management by statically computing the lifetimes of allocated heap objects [36, 22, 23]. However, as Java applications are written in such a way that they are not necessarily amenable to the static deallocation of objects, the deallocation sites discovered through these analyses are only a subset of all possible object deallocation sites. Due to this limitation, they still require garbage collection for the reclamation of dead objects. Moreover, to support explicit deallocation, this approach restricts the JVM runtime to use certain kinds of memory allocators that manage free objects using size-segregated linked lists. Unfortunately,

it has been shown that the performance of these memory allocators is poor compared to the latest memory allocators designed for Java applications.

The object recycle optimization this thesis presents uses both static analysis and dynamic profiling to overcome the shortcomings of the previous approaches. The static analysis we use is called reference uniqueness analysis [23]. In reference uniqueness analysis, the last use point of a variable that holds the only incoming reference to an object is identified as a safe-deallocation site. Using the analysis results along with the allocation sites in the codes, we perform a profiling to determine a set of allocation sites that are worthy of object recycle transformation (e.g., every object from the target allocation site has completely disjoint lifetimes). The object recycle optimization then transforms the selected allocation sites and their corresponding deallocation sites so that whenever an application needs to create an object from the selected sites, the application first consults the recycle cache to recycle a dead object.

CHAPTER VI

CONCLUSIONS AND FUTURE RESEARCH

6.1 Conclusions

Memory bloat is loosely defined as an excessive memory usage by an application during its execution. Due to the complexity of efficient memory management that developers have to deal with, memory bloat is pervasive and is often neglected in favor of lower application development cost. Unfortunately, when the bloat becomes severe, unwanted performance issues may occur due to its impact on memory management mechanisms and memory layout. For this reason, dealing with memory bloat is an important task for ensuring the optimal performance of software systems.

In light of this, this thesis investigates three predominant causes of memory bloat related performance problems and presents our solutions for the problems. The proposed suite of techniques tackles the memory bloat problem at three different levels: as a bug (memory leak), as a memory allocation/management efficiency problem (for TCMalloc), and as an optimization problem for object recycle (using both static and dynamic analyses). The following provides a summary of this thesis.

- `Chapter 2` addresses the dominant form of memory bloat that occurs due to the presence of memory leaks. Repeated occurrences of memory leaks cause the gradual exhaustion of system memory and eventually lead to serious performance degradation of production systems. To prevent the obvious consequences of memory leaks, this work presents a memory leak detection framework that relies on object behavior introspection. The introspective memory leak detection framework we present models behavioral changes of hypothetically leaked

objects in terms of their individual staleness and their allocation context coexistence patterns. Our framework observes the behaviors of allocated heap objects during the testing stage of application development and uses this knowledge to diagnose a presence of memory leaks during the production of a software. With this new leak detection method, applications should be able to attain significant memory bloat savings upon weeding out discovered memory leaks.

- `Chapter 3` provides a solution for a source of memory bloat related performance issues that arise due to the memory bloat prevention mechanism in multi-threaded memory allocators. When the bloat prevention mechanism is frequently triggered unnecessarily as an artifact of intensive memory allocations/deallocations, an application may experience a suboptimal performance. To address this, this work presents a feedback-directed tuning mechanism for TCMalloc, a widely used memory allocator for high performance systems. Our optimization technique tunes the thread cache management mechanism in TCMalloc to the memory allocation behavior of an application and reduces the management cost of the internal data structures in TCMalloc. With the proposed technique integrated into FDO in GCC, applications that use TCMalloc can be optimized to eliminate the performance impact due to the memory bloat inside the memory allocator.

- `Chapter 4` presents a solution for the memory bloat in Java applications that occurs due to performance unconscious designs and implementations. When an application uses an excessive amount of memory by creating more objects than are necessary, a negative performance impact such as a high garbage collection overhead may arise. To address this issue, this work presents an object recycle optimization technique for Java applications. Our technique uses a static analysis to figure out safe-deallocation sites of allocated objects and uses a dynamic

profiling to select the allocation sites for code transformation. By using the proposed optimization technique, applications that spend a significant amount of time creating objects can benefit from reduced memory bloat and achieve improved performance.

6.2 Future Research

The topics investigated in this thesis have possible extensions for future explorations.

6.2.1 Future Work for Introspective Memory Leak Detection

The introspective memory leak detection framework proposed in this thesis uses only object staleness and object coexistence patterns for diagnosing memory leaks. However, introspection of hypothetical behavioral change of heap objects does not need to be confined to these aspects of program execution. Any metric that may be impacted due to an occurrence of a memory leak can be considered as an input for model construction. For example, one candidate might be code coverage, i.e., objects may be live only during the execution of a certain code region.

However, care must be taken when selecting the metrics for behavior introspection. Usually, the profiling method for observing/measuring the metrics is based on sampling. Hence, one obvious restriction for the choice of the metrics is that a metric is not usable if it is highly affected by the estimation method in an uncontrollable way.

6.2.2 Future Work for Object Recycle Optimization

The object recycle optimization technique relies on the front-end static analysis for providing the safe-deallocation site information. The reference uniqueness analysis used in this thesis is a rather light-weight analysis in terms of the scope (i.e., once an object escapes into the heap, the analysis treats the lifetime of the escaped object

as intractable). Using more heavy-weight shape analysis may discover more safe-deallocation sites for the optimization. Moreover, applications may have polymorphic safe-deallocation sites that conditionally remove the last reference to an allocated object. If a program transformation can introduce a check to handle this case, even the polymorphic safe-deallocation sites may be handled by the object recycle optimization.

REFERENCES

[1] "Bash patch report." `http://ftp.gnu.org/gnu/bash/bash-4.0-patches/bash40-033`.

[2] "lighttpd - fly light." `http://www.lighttpd.net/`.

[3] "lighttpd issue tracker." `http://redmine.lighttpd.net/issues/1775`.

[4] "Lockless memory allocator." `http://locklessinc.com`.

[5] Bacon, D. F., Cheng, P., and Rajan, V. T., "Controlling fragmentation and space consumption in the metronome, a real-time garbage collector for java," in *Proceedings of the 2003 ACM SIGPLAN Conference on Language, Compiler, and Tool for Embedded Systems*, LCTES '03, (New York, NY, USA), pp. 81–92, ACM, 2003.

[6] Barrett, D. A. and Zorn, B. G., "Using lifetime predictors to improve memory allocation performance," in *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, PLDI '93, (New York, NY, USA), pp. 187–196, ACM, 1993.

[7] Bentley, P. J., "Climbing through complexity ceilings," *Network practices: new strategies in architecture and design. Princeton Architectural Press, New York*, 2007.

[8] Berger, E. D., McKinley, K. S., Blumofe, R. D., and Wilson, P. R., "Hoard: A scalable memory allocator for multithreaded applications," *ACM SIGPLAN Notices*, vol. 35, no. 11, pp. 117–128, 2000.

[9] Berger, E. D., Zorn, B. G., and McKinley, K. S., "Oopsla 2002: Reconsidering custom memory allocation," *SIGPLAN Not.*, vol. 48, pp. 46–57, July 2013.

[10] Bhattacharya, S., Nanda, M. G., Gopinath, K., and Gupta, M., "Reuse, recycle to de-bloat software," in *ECOOP 2011–Object-Oriented Programming*, pp. 408–432, Springer, 2011.

[11] Blackburn, S. M., Garner, R., Hoffman, C., Khan, A. M., McKinley, K. S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S. Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J. E. B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D., and Wiedermann, B., "The DaCapo benchmarks: Java benchmarking development and analysis," in *OOPSLA '06: Proceedings of the 21st annual ACM*

SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications, (New York, NY, USA), pp. 169–190, ACM Press, Oct. 2006.

[12] Blackburn, S. M., Cheng, P., and McKinley, K. S., "Myths and realities: The performance impact of garbage collection," in *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '04/Performance '04, (New York, NY, USA), pp. 25–36, ACM, 2004.

[13] Bond, M. and McKinley, K., "Tolerating memory leaks," in *Proceedings of the 23rd ACM SIGPLAN OOPSLA*, ACM, 2008.

[14] Bond, M. D. and McKinley, K., "Leak pruning," in *Proceeding of the 14th ASPLOS*, pp. 277–288, ACM, 2009.

[15] Bond, M. D. and McKinley, K. S., "Bell: bit-encoding online memory leak detection," in *Proc. of the 12th ASPLOS*, (New York, USA), 2006.

[16] Bruening, D. and Zhao, Q., "Practical memory checking with dr. memory," in *Proc. of the 9th CGO*, pp. 213–223, 2011.

[17] Burges, C. J., "A tutorial on support vector machines for pattern recognition," *Data mining and knowledge discovery*, vol. 2, no. 2, pp. 121–167, 1998.

[18] Catanzaro, B., Sundaram, N., and Keutzer, K., "Fast support vector machine training and classification on graphics processors," in *Proceedings of the 25th international conference on Machine learning*, ICML '08, (New York, NY, USA), pp. 104–111, ACM, 2008.

[19] Chang, C.-C. and Lin, C.-J., "LIBSVM: A library for support vector machines," *ACM Transactions on Intelligent Systems and Technology*, vol. 2, pp. 27:1–27:27, 2011. Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`.

[20] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E., "Bigtable: A distributed storage system for structured data," *ACM Transactions on Computer Systems*, vol. 26, no. 2, 2008.

[21] Cherem, S., Princehouse, L., and Rugina, R., "Practical memory leak detection using guarded value-flow analysis," in *Proc. of 28th PLDI'07*.

[22] Cherem, S. and Rugina, R., "A practical escape and effect analysis for building lightweight method summaries," in *Compiler Construction*, pp. 172–186, Springer, 2007.

[23] Cherem, S. and Rugina, R., "Uniqueness inference for compile-time object deallocation," in *Proceedings of the 6th international symposium on Memory management*, pp. 117–128, ACM, 2007.

[24] Chilimbi, T. M. and Hauswirth, M., "Low-overhead memory leak detection using adaptive statistical profiling," in *Proc. of 11th ASPLOS'04*.

[25] Chis, A. E., Mitchell, N., Schonberg, E., Sevitsky, G., OSullivan, P., Parsons, T., and Murphy, J., "Patterns of memory inefficiency," in *ECOOP 2011–Object-Oriented Programming*, pp. 383–407, Springer, 2011.

[26] Clause, J. and Orso, A., "Leakpoint: pinpointing the causes of memory leaks," in *Proc. of the 32nd ICSE*, (New York, NY, USA), 2010.

[27] Cortes, C. and Vapnik, V., "Support-vector networks," *Machine Learning*, vol. 20, no. 3, pp. 273–297, 1995.

[28] Cotroneo, D., Natella, R., and Pietrantuono, R., "Predicting aging-related bugs using software complexity metrics," *Performance Evaluation*, vol. 70, no. 3, pp. 163–178, 2013.

[29] Dufour, B., Ryder, B. G., and Sevitsky, G., "Blended analysis for performance understanding of framework-based applications," in *Proceedings of the 2007 international symposium on Software testing and analysis*, pp. 118–128, ACM, 2007.

[30] Evans, J., "jemalloc." `http://www.canonware.com/jemalloc`.

[31] Evans, J., "Scalable memory allocation using jemalloc," 2011. `http://www.facebook.com/notes/facebook-engineering/scalable-memory-allocation-using-jemalloc/480222803919`.

[32] Ghemawat, S. and Menage, P., "Tcmalloc : Thread-caching malloc," `http://goog-perftools.sourceforge.net/doc/tcmalloc.html`.

[33] Gidenstam, A., Papatriantafilou, M., and Tsigas, P., "Allocating memory in a lock-free manner," in *Algorithms–ESA 2005*, pp. 329–342, Springer, 2005.

[34] Graf, H. P., Cosatto, E., Bottou, L., Dourdanovic, I., and Vapnik, V., "Parallel support vector machines: The cascade svm," in *Advances in neural information processing systems*, pp. 521–528, 2004.

[35] Grunwald, D. and Zorn, B., "Customalloc: Efficient synthesized memory allocators," *Software: Practice and Experience*, vol. 23, no. 8, pp. 851–869, 1993.

[36] Guyer, S. Z., McKinley, K. S., and Frampton, D., "Free-me: a static analysis for automatic individual object reclamation," *ACM SIGPLAN Notices*, vol. 41, no. 6, pp. 364–375, 2006.

[37] Heine, D. L. and Lam, M. S., "A practical flow- and context-sensitive c/c++ memory leak detector," in *Proc. of the 23rd PLDI*, 2003.

[38] Hsu, C.-W., Chang, C.-C., and Lin, C.-J., "A practical guide to support vector classification," 2010.

[39] Inoue, H., Stefanovic, D., and Forrest, S., "On the prediction of java object lifetimes," *Computers, IEEE Transactions on*, vol. 55, pp. 880–892, July 2006.

[40] Jung, C. and Clark, N., "Ddt: design and evaluation of a dynamic program analysis for optimizing data structure usage," in *MICRO 42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, (New York, NY, USA), pp. 56–66, ACM, 2009.

[41] Jung, C., Woo, D.-K., Kim, K., and Lim, S.-S., "Performance characterization of prelinking and preloading for embedded systems," in *Proceedings of the 7th ACM & IEEE international conference on Embedded software*, EMSOFT '07, (New York, NY, USA), pp. 213–220, ACM, 2007.

[42] Jung, Y. and Yi, K., "Practical memory leak detector based on parameterized procedural summaries," in *Proc. of the 7th ISMM*, 2008.

[43] Krishnan, M. R., "Heap: Pleasures and pains," *Microsoft Developer Newsletter*, 1999.

[44] Lattner, C. and Adve, V., "LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation," pp. 75–86, 2004.

[45] Lim, W., Park, S., and Han, H., "Memory leak detection with context awareness," in *Proceedings of the 2012 ACM Research in Applied Computation Symposium*, RACS '12, (New York, NY, USA), pp. 276–281, ACM, 2012.

[46] Michael, M. M., "Scalable lock-free dynamic memory allocation," in *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, PLDI '04, (New York, NY, USA), pp. 35–46, ACM, 2004.

[47] Mitchell, N., Schonberg, E., and Sevitsky, G., "Four trends leading to java runtime bloat," *IEEE Software*, vol. 27, pp. 56–63, 2010.

[48] Mitchell, N. and Sevitsky, G., "The causes of bloat, the limits of health," in *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, OOPSLA '07, (New York, NY, USA), pp. 245–260, 2007.

[49] Myhrvold, N., "The next fifty years of software," in *ACM97 Conference*, 1997.

[50] Novark, G., Berger, E. D., and Zorn, B. G., "Efficiently and precisely locating memory leaks and bloat," in *Proc. of the 30th PLDI*, 2009.

[51] Oaks, S., *Java Performance: The Definitive Guide*. " O'Reilly Media, Inc.", 2014.

[52] Orlovich, M. and Rugina, R., "Memory leak analysis by contradiction," in *In Proceedings of the 13th International Static Analysis Symposium*, pp. 405–424, 2006.

[53] Qin, F., Lu, S., and Zhou, Y., "Safemem: Exploiting ecc-memory for detecting memory leaks and memory corruption during production runs," in *Proc. of the 11th HPCA*, 2005.

[54] Schaller, R. R., "Moore's law: past, present and future," *Spectrum, IEEE*, vol. 34, no. 6, pp. 52–59, 1997.

[55] Shacham, O., Vechev, M., and Yahav, E., "Chameleon: adaptive selection of collections," in *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, pp. 408–418, 2009.

[56] Shankar, A., Arnold, M., and Bodik, R., "Jolt: lightweight dynamic analysis and removal of object churn," in *ACM Sigplan Notices*, vol. 43, pp. 127–142, ACM, 2008.

[57] Shirazi, J., *Java performance tuning.* " O'Reilly Media, Inc.", 2003.

[58] Smith, M. D., "Overcoming the challenges to feedback-directed optimization (keynote talk)," in *Proceedings of the ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization*, DYNAMO '00, (New York, NY, USA), pp. 1–11, ACM, 2000.

[59] Sui, Y., Ye, D., and Xue, J., "Static memory leak detection using full-sparse value-flow analysis," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, (New York, NY, USA), pp. 254–264, ACM, 2012.

[60] Tang, Y., Gao, Q., and Qin, F., "Leaksurvivor: towards safely tolerating memory leaks for garbage-collected languages," in *Proc. of USENIX 2008 Annual Technical Conference*.

[61] Weisner, R. C., "How memory allocation affects performance in multithreaded programs," *System News for Sun Users*, 2012.

[62] Whittaker, J., *How to Break Software Security.* Addision Wesley.

[63] Williams, A., "Amazon web services outage caused by memory leak and failure in monitoring alarm." `http://techcrunch.com/2012/10/27/amazon-web-services-outage-caused-by-memory-leak-and-failure-in-monitoring-alarm/`, 10 2012.

[64] Xie, Y. and Aiken, A., "Context- and path-sensitive memory leak detection," in *Proc. of ESEC/FSE 2005*, ACM Press, 2005.

[65] Xu, G., "Finding reusable data structures," in *ACM SIGPLAN Notices*, vol. 47, pp. 1017–1034, ACM, 2012.

[66] Xu, G., "Resurrector: A tunable object lifetime profiling technique for optimizing real-world programs," in *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*, pp. 111–130, ACM, 2013.

[67] Xu, G., Arnold, M., Mitchell, N., Rountev, A., and Sevitsky, G., "Go with the flow: profiling copies to find runtime bloat," in *ACM Sigplan Notices*, vol. 44, pp. 419–430, ACM, 2009.

[68] Xu, G., Mitchell, N., Arnold, M., Rountev, A., and Sevitsky, G., "Software bloat analysis: finding, removing, and preventing performance problems in modern large-scale object-oriented applications," in *Proceedings of the FSE/SDP workshop on Future of software engineering research*, FoSER '10, (New York, NY, USA), pp. 421–426, ACM, 2010.

[69] Xu, G., Mitchell, N., Arnold, M., Rountev, A., and Sevitsky, G., "Software bloat analysis: Finding, removing, and preventing performance problems in modern large-scale object-oriented applications," in *Proceedings of the FSE/SDP workshop on Future of software engineering research*, pp. 421–426, ACM, 2010.